Visual personalizado para Mobile: Explore a criação de estilos

Clube Delphi

Edição 158 :: R$ 14,90

DEVMEDIA

Integração Contínua
Otimize os builds de seu software

Mini curso: EntityDAC
Uma alternativa ao TMS Aurelius

Seu software pode estar devendo
Entenda o que é isso e como corrigir

DELPHI XE6

Fique por dentro das novidades

ISSN 1517990-7

CURSOS ONLINE

Clube Delphi

A Revista Clube Delphi oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

CONHEÇA ALGUNS DOS CURSOS

- Curso de Multicamadas com Delphi e DataSnap
- Delphi para Iniciantes
- Criando componente Boleto em Delphi
- Loja Virtual em Delphi Prism

Para mais informações :

www.devmedia.com.br/cursos/delphi

(21) 3382-5038

Clube Delphi

158ª Edição 2014 ISSN 1517990-7 Impresso no Brasil

Editor Geral

Paulo Quicoli (pauloquicoli@gmail.com)

Equipe Editorial

Guinther Pauli (guintherpauli@gmail.com)

Fabrício Hissao Kawata (fabricio.kawata@bol.com.br)

Giuliano Scombatti Pinto (giuliano@sygnux.com.br)

Daniel Sobrinho Laporte (daniel.laporte@gmail.com)

Jornalista Responsável

Kaline Dolabella - JP24185

Consultor Técnico

Davi Costa (davigc_08@hotmail.com)

Diagramação e Capa

Romulo Araujo

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Paulo Quicoli - Editor da Revista
pauloquicoli@gmail.com

Paulo Quicoli - **Editor Geral**

pauloquicoli@gmail.com
quicoli.wordpress.com
twitter.com/pauloquicoli

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia: **www.devmedia.com.br/mvp**

DEVMEDIA

Sumário

Conteúdo sobre Novidades

04 – Delphi XE6: Novidades da versão
[*Fabricio Hissao Kawata*]

Conteúdo sobre Engenharia de Software

14 – SVN e Jenkins: Integração contínua
[*Diego Garcia*]

Artigo no estilo Curso

24 – EntityDAC: Ferramenta de Mapeamento Objeto-Relacional – Parte 1
[*Fabrício Hissao Kawata*]

Conteúdo sobre Boas Práticas, Artigo no estilo Mentoring

35 – Metodologia Ágil no Delphi XE6
[*Guinther Pauli*]

Artigo no estilo Solução Completa

46 – FireMonkey Styles: Personalização de ListBox em aplicações mobile
[*Jousival Costa Vilela*]

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Novidades do Delphi XE6

Overview das novidades desta versão

Passados menos de um ano após o lançamento de sua versão anterior (XE5), o Delphi é novamente trazido à tona com a projeção de seu mais recente release, o Delphi XE6. Sem contar com grandes incorporações, esta nova versão do produto prima pela solidificação dos principais recursos surgidos em suas últimas versões, tal como a capacidade de produção de aplicações móveis para as plataformas líderes de mercado – iOS e Android. Além disso, questões como o desenvolvimento baseado em recursos plenamente visuais, acesso à API da plataforma e velocidade são novamente salientadas, agora num cenário que envolve diferentes dispositivos alvo, tais como PCs, Tablets e Smartphones.

A seguir são então apresentadas as principais novidades desta versão XE6, agrupadas de acordo com sua área de interesse relacionada.

IDE

Basta uma primeira olhada no ambiente de desenvolvimento do Delphi XE6 para que se perceba uma grande, porém sutil, mudança estética de seu layout. Pondo isto em palavras, significa dizer que a identidade visual da ferramenta, estilizada ao longo de sua linha de vida, agora ganha novos ares. Isso se reflete num aspecto mais moderno, com seus ícones mais "chapados", não apresentando o realce habitual. Adicionalmente, diversos ícones, além de reestilizados, também sofreram atualizações, no que diz respeito à sua imagem de representação. Toda esta nova conjuntura é então mostrada na **Figura 1**.

Google Glass

De forma breve, o Google Glass pode ser definido como sendo um acessório provido pelo Google, para uso pessoal, em forma de óculos, que possibilita a interação de seus utilizadores (usuários) com diversos conteúdos em realidade aumentada. Internamente, o dispositivo faz uso da plataforma Android, o que dá margens para que os desenvolvedores desse segmento explorem a tecnologia. Em vista disso, com seu recente suporte a esse sistema operacional, o Delphi XE6 traz agora um novo design de aplicação, mais especificamente um novo Form Designer para projetos de aplicações móveis (Mobile Form Designer) voltado ao contexto do Google Glass. De forma ilustrativa, a **Figura 2** mostra esta nova opção habilitada em um projeto FireMonkey Mobile Application.

Fique por Dentro

Este artigo se fundamenta na apresentação geral da recém-lançada versão do Delphi, a XE6. Sendo assim, serão vistos os principais pontos de destaque que integram este release, que se refletem tanto em novos recursos adicionados quanto na melhoria de outros anteriormente existentes. Assim como já é de praxe na trajetória da ferramenta, toda nova versão lançada traz como chamariz um determinado tópico "chave", que está associado naturalmente ao âmbito de suas principais novidades. Logo, tal como o "Desenvolvimento Móvel para Android" estava associado à versão XE5, o Delphi XE6 traz como tema principal a "construção rápida de aplicações para múltiplas plataformas". Tudo isso se traduz então no slogan comercial "O caminho mais rápido para construir aplicativos nativos para Windows, Mac, iOS e Android", numa clara busca pela estabilização concreta dos recursos incorporados mais recentemente nas últimas versões do produto.

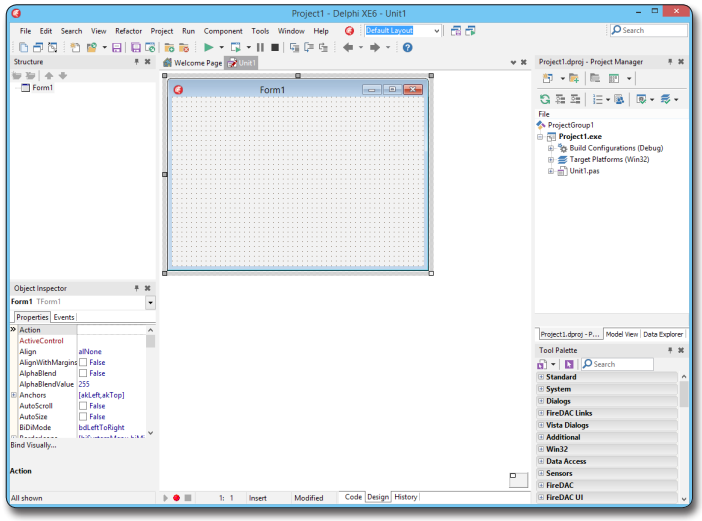


Figura 1. IDE do Delphi XE6

Deployment Manager

Também pautado como uma das adições recentes do Delphi, o Deployment Manager, conforme seu próprio nome sugere, é uma estrutura de gerenciamento de deploys de aplicações para as diferentes plataformas de destino suportadas atualmente pela ferramenta. De forma prática, através de seu uso o desenvolvedor é capaz de visualizar, adicionar, editar ou excluir os arquivos que farão parte do processo de deploy. Como exemplo, tendo como base um projeto do

tipo FireMonkey Mobile para a construção de uma aplicação móvel de banco de dados para Android, é através do Deployment Manager que se dá a inclusão do driver do SGBD específico a ser implantado no dispositivo alvo. Naturalmente, tal ação pode ser considerada como essencial para o funcionamento da aplicação.

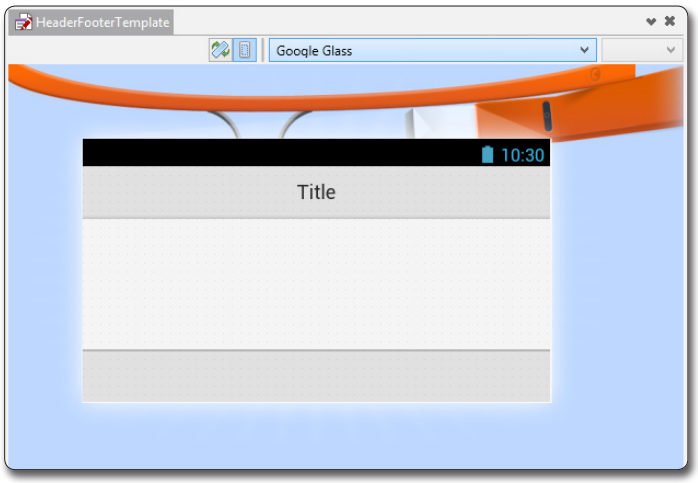


Figura 2. Mobile Form Designer – Google Glass

Dada sua importância, o Deployment Manager ganha duas novas opções nesta nova versão do Delphi. A primeira delas é então apresentada sob o título de "Overwrite", cuja finalidade é controlar a substituição (overwriting) indevida e/ou indesejada, no momento de deploy, de arquivos já presentes no dispositivo alvo. Para tal, possibilita a escolha e indicação pontual dos itens que deverão (Always) ou não (Never) serem sobrescritos no processo de implantação. Em razão disso, a opção trabalha com duas alternativas de valores – Always e Never – sendo a primeira seu valor padrão. A fim de ilustrar o que foi dito, a seguir são mostrados, em destaque na **Figura 3**, o botão de ação e a coluna informativa, relacionados à opção Overwrite, presentes no Deployment Manager desta nova versão do IDE.

Em complemento, a outra nova opção disponibilizada neste cenário é denominada "Keep added files" e prevê a manutenção de todo e qualquer arquivo adicionado manualmente ao projeto, no que diz respeito ao seu deploy. De forma específica, tomando como base projetos Mobile para Android, tal recurso surge pelo fato de que este tipo de projeto, por padrão, já contempla um agregado de arquivos tidos como essenciais para a aplicação e que por este motivo farão parte do processo de implantação. Prova disso pode ser vista na própria imagem anterior, que foi retirada de um projeto nativo FireMonkey Mobile Application, onde podem ser vistos os arquivos adicionados por padrão à lista do Deployment Manager. Assim, a nova opção "Keep added files" se incumbe de garantir todo e qualquer arquivo adicionado pelo desenvolvedor à essa lista. Com a intenção de preservar a integridade dos projetos, o Delphi já traz então esta opção habilitada, conforme mostra a **Figura 4**.

Esta imagem faz menção à janela "Please specify the action", que é trazida à tona por meio do acionamento do botão Revert do Deployment Manager, que permite se reverter as configurações de deploy para as opções padrão, descartando todas as mudanças feitas de forma manual até então. Mediante isso, a opção "Keep added files", uma vez marcada, preserva os eventuais arquivos adicionados à lista de deploy e que, portanto, não fazem parte da configuração padrão.

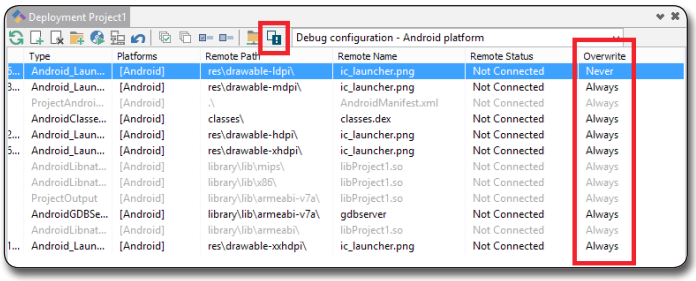


Figura 3. Deployment Manager – Opção "Overwrite"

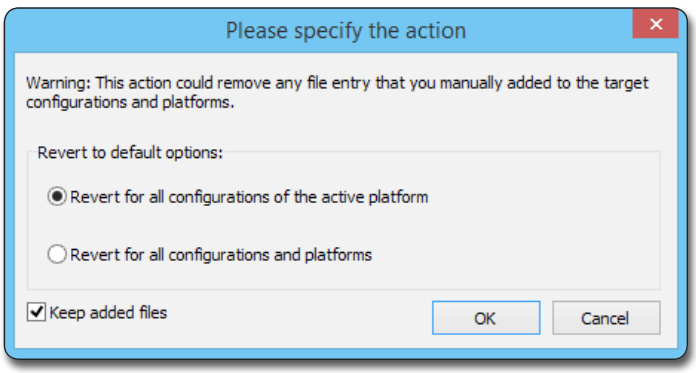


Figura 4. Deployment Manager – Opção "Keep added files"

Tool Options – SDK Manager

Em sua versão anterior (XE5) o Delphi teve como acréscimo ao seu contexto um novo segmento de projetos, agora relacionado à construção de aplicações para dispositivos móveis relacionados à plataforma Android. Isso fez com que diversos novos aspectos também fossem adicionados ao IDE, a fim de adequá-lo a todo este novo cenário. Dentre estas "inovações" está a opção SDK Manager, que passou então a integrar a família de itens disponibilizados nas opções da ferramenta (Tools | Options). Em síntese, sua função básica é prover os meios necessários à configuração das propriedades relacionadas ao contexto do desenvolvimento Android, que é composto essencialmente por três elementos: SDK, NDK e Java. Na prática, essa "configuração" é refletida em campos informativos, relacionados ao caminho físico (path) de localização de determinados arquivos de recurso pertencentes a cada um destes elementos. Com esse tipo de abertura, o Delphi consegue então prover ao desenvolvedor, meios suficientes para se operar com diferentes versões de Android SDK a partir de um mesmo ambiente de desenvolvimento.

No XE5 tais propriedades se agrupavam num único lugar físico, o que provia uma certa confusão, além de uma natural poluição visual ao primeiro contato. Por esta razão para fins de organização

e clareza, nesta nova versão do IDE as propriedades de configuração de cada um dos três elementos são disponibilizadas em guias separadas, conforme mostra a **Figura 5**.

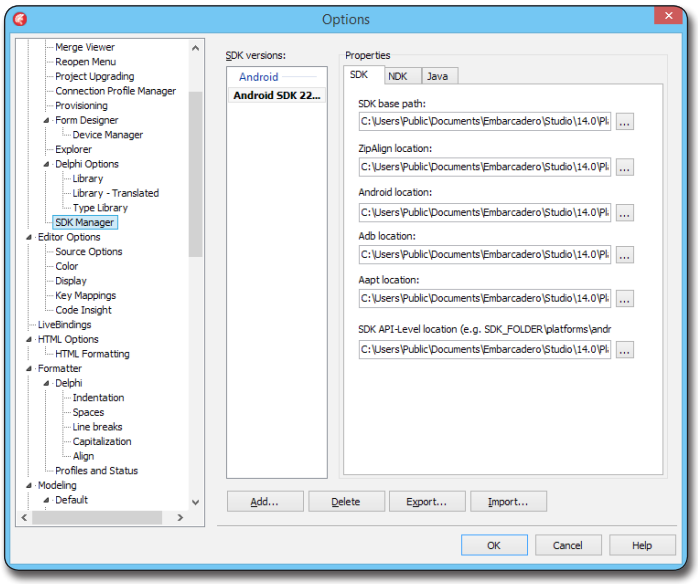


Figura 5. Options – SDK Manager

Project Options (para projetos FireMonkey Mobile Application)

As opções de projeto no novo IDE do Delphi ganham adições pontuais, direcionadas a tipos de projetos específicos, neste caso, para projetos do tipo FireMonkey Mobile Application. Logo, a primeira novidade apresenta-se sob o nome de Orientation, e caracteriza um novo sub-nó da opção de projeto Application (Project | Options | Application). Tal como a própria denominação propõe, sua funcionalidade é habilitar a customização da orientação do aplicativo Mobile que está sendo construído, possibilitando a definição explícita dos tipos de Orientation que a aplicação irá suportar. Conforme pode ser visto na **Figura 6**, no release atual, são quatro as opções disponíveis: Portrait (retrato), Upside Down (cabeça para baixo), Landscape (paisagem) home right e home left.

Ainda seguindo por este contexto, outra novidade apresentada nesta versão está associada à opção de projeto Version Info, que aloca uma porção de campos configuráveis relacionados ao informativo da versão da aplicação Mobile em questão. Para o desenvolvedor, cada informação dessa é disponibilizada na forma de entradas Key/Value (chave-valor), determinando configurações como o código e o nome da versão, nome do pacote do módulo, tema utilizado, entre outras informações. Adicionalmente, agora no XE6 um novo item informativo, denominado hardwareAccelerated (**Figura 7**), é acrescido a este agrupamento. De valor booleano (true/false), quando definido como True, indica a habilitação do recurso de aceleração de hardware para a aplicação inteira.

Data Explorer

Data Explorer é o nome dado a uma janela localizada nativamente no canto superior direito do IDE do Delphi, apresentada na forma de guia, juntamente dos tradicionais Project Manager e Model View. Sua função é prover acessibilidade a fontes de dados externas, mais especificamente Servidores de Bancos de Dados e Servidores de Aplicação DataSnap, a partir do próprio ambiente de desenvolvimento. A partir disso, sua estrutura apresenta uma extensa lista de tipos de fontes disponíveis, incluindo os tradicionais SGBDs utilizados na construção de aplicações Delphi de banco de dados, tais como Firebird, InterBase, MySQL e Oracle, o que permite a rápida criação e gerenciamento de conexões a cada uma delas. Na prática, o desenvolvedor será então capaz de navegar entre os objetos do schema, visualizando e manipulando elementos do tipo Table, View, Stored Procedure, Field, Index e Trigger. Adicionalmente, o simples arrastar de uma conexão definida no Data Explorer para o projeto da aplicação, faz com que os componentes de acesso a dados condizentes e necessários sejam automaticamente adicionados, já totalmente configurados.

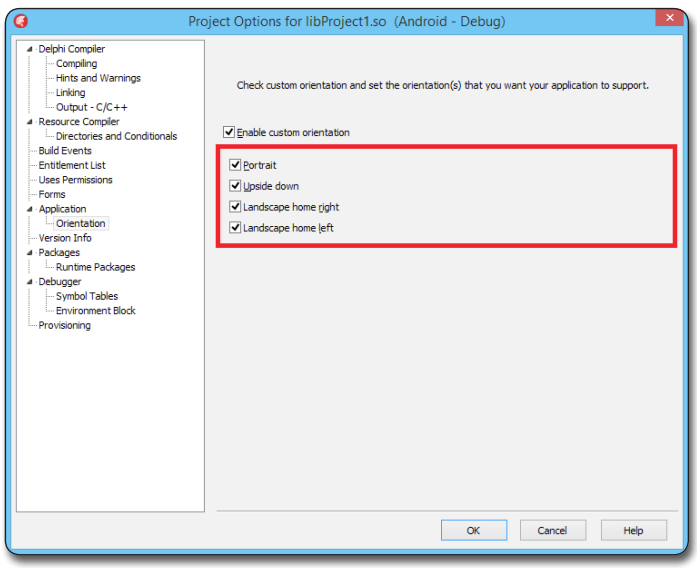


Figura 6. Project Options – Application – Orientation

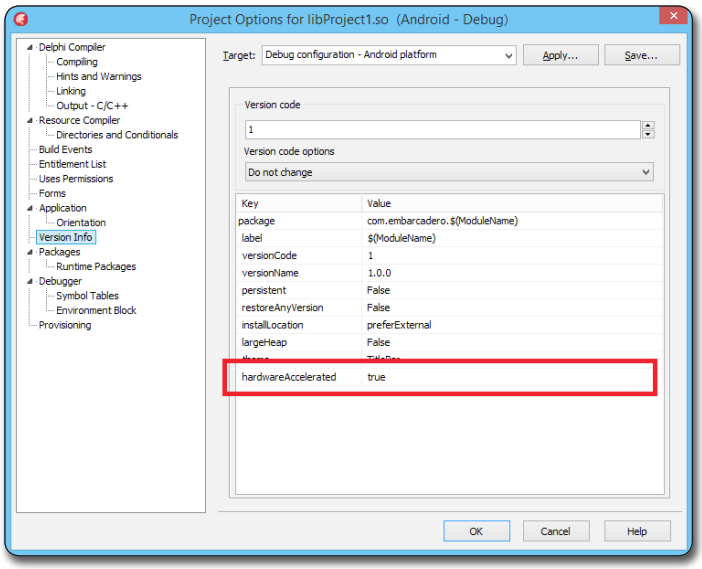


Figura 7. Project Options – Version Info – Hardware Accelerated

Nas versões anteriores do IDE o Data Explorer era uma estrutura exclusivamente direcionada ao dbExpress, principal opção nativa em termos de componentes de acesso a dados, para o desenvolvimento de aplicações de banco de dados. Justificada pela própria trajetória histórica da ferramenta, essa verdade foi plenamente mantida até o release XE4. Isto porque, já no Delphi XE5 o posto de "principal tecnologia de acesso a dados" passou a ser dividido e até comercialmente sobrepujado pelo FireDAC, que remete a uma biblioteca de componentes de terceiros, adquirida pela Embarcadero, e recentemente incorporada ao contexto da ferramenta. Diante disso, esta nova versão do IDE traz agora um Data Explorer totalmente remodelado, com a adição do FireDAC ao seu âmbito. Em termos práticos, este novo aspecto dá origem a um novo nó estrutural e de recursos, conforme ilustrado na **Figura 8**.

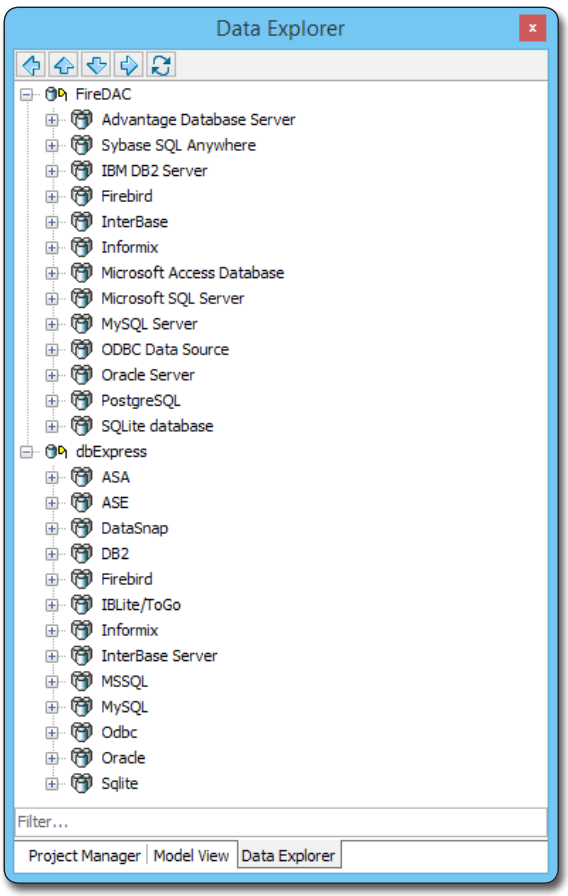


Figura 8. Data Explorer remodelado nesta versão XE6

Ainda com base na imagem, outra constatação imediata com relação ao FireDAC se dá pela existência de uma extensa lista de opções de acesso a diferentes tipos de fontes de dados, tal como no dbExpress. Dentre esta listagem estão servidores tradicionais como ASA (Advantage Database Server), MySQL, PostgreSQL, Oracle e Microsoft SQL Server, como também o opcional ODBC Data Source, fator exclusivo do FireDAC, que provê a conexão direta com fontes ODBC.

VCL

O cenário atual do Delphi expõe de maneira clara a segregação estabelecida no contexto operacional da ferramenta, por meio da presença de duas grandes segmentações em seu âmbito de desenvolvimento: VCL e FMX. A primeira, historicamente tradicional e muito bem aceita e utilizada pela comunidade, passa agora a dividir os holofotes com a segunda, que se incumbe em atender toda a parte multiplataforma que a ferramenta passa a exigir. Logo, mediante os lançamentos mais recentes do produto, é possível afirmar que grande parte das novidades surgidas nos tempos atuais estão naturalmente direcionadas ao mais recente framework de desenvolvimento da ferramenta, o FireMonkey (FMX), que passa por um período de maturidade e estabilização. Tomando como base a versão XE5, praticamente zero foram as melhorias ou mudanças ocorridas na VCL, justificada principalmente em razão do apelo comercial Mobile/Multiplataforma atrelado ao produto na ocasião. Não obstante, agora na versão XE6, a Embarcadero novamente volta seus olhos também à VCL, proporcionando melhorias pontuais nesta sua segmentação.

Componentes

Pontualmente falando, a primeira novidade da VCL se dá pela extensão da cobertura dos componentes sensoriais, até então

direcionadas exclusivamente à FMX, por meio do namespace FMX.Sensors. Para isso, toda infra de elementos como TLocationSensor, TMotionSensor e TOrientationSensor está agora centralizada em uma nova unit condizente ao contexto da RTL, denominada System.Sensors.Components. Como consequência, esta mudança estrutural faz com que os mesmos Sensor Components estejam disponíveis tanto em construções VCL quanto FMX. Em complemento, a **Figura 9** exibe o grupo Sensors, disponibilizado na Tool Palette.

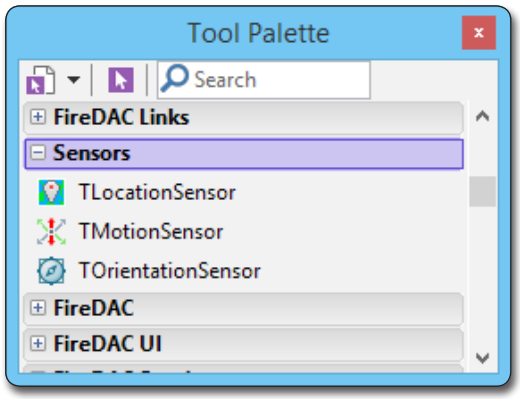


Figura 9. Tool Palette – Grupo Sensors agora disponível em projetos VCL

Ainda sobre os componentes da VCL, essa nova versão do Delphi também agrega melhorias ao componente TWebBrowser, muito utilizado na construção de aplicativos Desktop que, de alguma forma, realizam interações com páginas Web. Para isso, em tempo de execução (runtime), a estrutura interna do componente se utiliza do próprio navegador da plataforma, o Internet Explorer, para sua navegabilidade. Logo, agora no XE6, o referido componente é atualizado, sendo direcionado à mais recente versão deste browser.

Estilos

Apresentado como uma das grandes novidades da versão XE2, o recurso de VCL Styles provido pelo Delphi pode ser brevemente comparado à tecnologia de Skin, muito utilizado pela comunidade para incrementar o visual de suas aplicações. Devido a sua facilidade de uso e do atrativo proporcionado ao usuário final, esta capacidade de estilização nativa oferecida pela ferramenta é plenamente adotada pelos desenvolvedores que se utilizam das versões mais recentes do IDE. Em função disso, o Delphi XE6 traz agora novas adições a este contexto, por meio de dois novos estilos, apresentados como TabletDark e Glossy. Sem deixar de mencionar, o primeiro pode ser considerado um estilo "especial", uma vez que seu direcionamento é voltado a aplicações Windows baseadas em interface touch, tal qual para tablets.

Melhorias para Windows 7 e Windows 8

Assim como já é de conhecimento de todos, historicamente o Delphi foi concebido como sendo uma ferramenta de desenvolvimento de aplicativos Windows. Sendo assim, durante toda sua trajetória, incluindo o presente momento, grande parte de seu poderio de habilidades e reconhecimento popular está relacionado a essa sua característica. Diante da natural evolução da própria plataforma a que atende, o Delphi busca acompanhar cada inovação surgida, proporcionando sempre o que há de melhor e mais novo em meio ao seu cenário de desenvolvimento de aplicações. Prova disso é que em razão das recentes adições de novos paradigmas visuais e de comportamento, impostos pelas recentes versões 7 e 8 do Windows, o Delphi XE6 traz então melhorias pontuais para Taskbar Buttons, que são os botões atrelados à barra de tarefas dos referidos sistemas operacionais. Tais novidades são providas por meio da nova unit Vcl.Taskbar, que estruturalmente centraliza os diversos elementos deste contexto, incluindo o componente TTaskbar (disponível no grupo Win32 da Tool Palette). A seguir são elencadas algumas das melhorias apresentadas:

- Trabalhar com sobreposição de imagens no ícone da aplicação na Taskbar;
- Utilizar barra de progresso nos botões da Taskbar;
- Controlar a atualização e o estado de cores da barra de progresso relacionada;
- Operar com múltiplos botões na visualização da miniatura do aplicativo na Taskbar;
- Especificamente para aplicações MDI, cada janela filha (child) é automaticamente adicionada ao preview da aplicação na Taskbar, formalizando uma nova janela. Na prática, este comportamento se assemelha ao do Internet Explorer, que disponibiliza uma mini-janela de preview para cada guia que esteja aberta.

FireMonkey

Assim como a VCL, o FireMonkey conta também com diversas apresentações em seus mais variados segmentos, neste novo release do Delphi. Alterações de nomenclaturas de seus elementos, adições de novos componentes, atualizações pontuais em componentes existentes e descontinuidade de outros são alguns exemplos das mudanças ocorridas no framework, trazidas nesta nova versão do Delphi. Obviamente, em meio a tudo isso, inúmeras são as alterações de menor porte, que passam despercebidas ao desenvolvedor, muito em razão de seu baixo impacto no desenvolvimento efetivo. Por esta razão a seguir são elencadas algumas das principais novidades relacionadas a FireMonkey nesta versão.

Componentes

Em linhas gerais, as mudanças de maior impacto ocorridas no cenário FMX estão relacionadas à linha de componentes. Como exemplo inicial é possível citar o elemento visual TTabControl, que nesta versão ganha o acréscimo de novos métodos, mais precisamente cinco – Add, Insert, Delete, Next e Previous – que fundamentam o controle e manipulação de guias (tabs). Adicionalmente, conforme mencionado, há ainda elementos disponíveis até a versão XE5 e que agora no Delphi XE6 foram descontinuados, sendo então considerados "deprecated" perante o framework. Este é o caso do componente visual TCalendarEdit, declarado até então na unit FMX.DateTimeCtrls e que representa uma caixa de texto

simples e editável para apontamento de data, cuja seleção ocorre por meio de um controle drop down TCalendar. Neste caso em específico, a descontinuidade citada pode então ser considerada uma substituição, dado o surgimento de dois novos componentes relacionados a este panorama, são eles TDateEdit e TTimeEdit, disponibilizados no grupo Additional da Tool Palette (**Figura 10**). Em termos didáticos, vale ressaltar que as classes de ambos também se encontram sob o escopo FMX.DateTimeCtrls, tal qual o do anterior TCalendarEdit. Já olhando pelo aspecto prático, a própria nomenclatura de cada um sugere sua funcionalidade. Logo, o componente TDateEdit quando utilizado, reflete-se numa caixa de texto simples para controle específico de data que é selecionada por meio de um "date picker". Da mesma forma, o componente TTimeEdit provê ao desenvolvedor uma caixa de texto para controle específico de tempo, selecionado por meio de um "time picker".

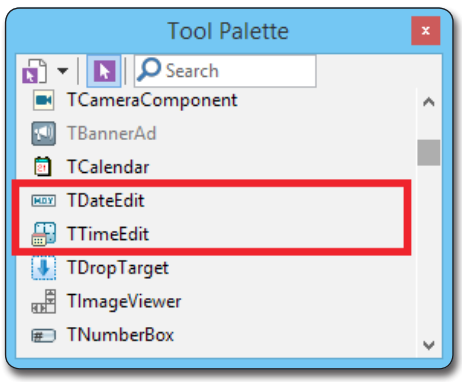


Figura 10. Novos componentes FMX: TDateEdit e TTimeEdit

Em geral, além de "exclusões" e novas adições, a biblioteca do FireMonkey também apresenta nesta versão alterações pontuais em alguns de seus componentes existentes. Nesta linha estão três elementos bastante tradicionais em meio ao desenvolvimento FMX – TButton, TSpeedButton e TToolBar. Neste release, todos eles agora contam com uma nova propriedade, denominada TintColor, do tipo TAlphaColor, que foi adicionada às suas classes com o intuito de prover ao desenvolvedor final a aplicação de tons (tonalidade) nos respectivos controles visuais. Logo, tomando como exemplo o controle TButton, o referido aspecto está relacionado à sua cor de fundo (background color).

Ainda nesta linha de "atualizações", TButton e TSpeedButton apresentam adicionalmente outra nova propriedade condizente a esta determinação de tonalidade, denominada IconTintColor, que permite o desenvolvedor especificar a cor do ícone destes controles, quando da utilização de estilos (StyleLookup).

Componentes exclusivos para iOS e Android

Ainda em se tratando de componentes, agora no XE6, alguns novos elementos surgem em meio à biblioteca da FMX, voltados exclusivamente à construção de aplicativos móveis. Dentre estes, o primeiro a ser citado apresenta-se sob o nome de TInAppPurchase, disponibilizado no grupo Services da Tool Palette (**Figura 11**). Seu direcionamento está estreitamente relacionado a um tipo de serviço atrelado às aplicações Mobile, que diz respeito à venda e pagamento de conteúdo digital por meio das próprias aplicações para iOS e Android. Sem deixar de citar, no meio, este tipo de atividade é tratado como In-App Payment Services. Por conseguinte, a função principal do componente se resume em abstrair do desenvolvedor todas as particularidades relativas a este processo, tornando natural a adição destes tipos de recursos nas aplicações FireMonkey Mobile construídas nesta versão da ferramenta.

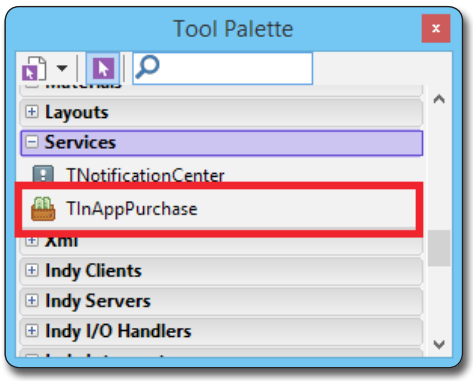


Figura 11. Grupo Services – Novo componente TInAppPurchase

Seguindo por esta linha, TBannerAd (**Figura 12**) é outro componente adicionado nesta conjuntura, cuja função é abstrair as nuances que envolvem os serviços de publicidade atrelados à plataforma da aplicação Mobile. No popular, este tipo de serviço é então conhecido como Advertising Service e de forma prática, possibilita que o desenvolvedor exiba anúncios em suas aplicações Android e iOS a fim de angariar renda, direta ou indiretamente. Através do uso deste novo elemento visual, o desenvolvedor Delphi é capaz agora de adicionar em suas aplicações FireMonkey Mobile o devido suporte aos serviços de publicidade relacionados, numa abordagem totalmente RAD, pelo simples arrastar-e-soltar de um componente em tela e posterior configuração. Sem deixar de mencionar, nos bastidores, cada plataforma trabalha com um tipo de serviço próprio, tal qual iAd (ver **BOX 1**) para iOS e AdMob (ver **BOX 1**) para Android. Funcionalmente, este tipo de atividade tende a funcionar como um incentivo a mais para a construção dos mais variados tipos de aplicativos a serem distribuídos ao grande público consumidor.

BOX 1. iAd e AdMob

De forma sucinta, iAd é o nome dado a uma plataforma de serviços de publicidade concebida pela Apple, direcionada ao seu segmento de dispositivos móveis – iPhone, iPad e iPod Touch. Na prática, ela possibilita que desenvolvedores insiram anúncios comerciais de terceiros diretamente em suas aplicações, gerando e obtendo renda. Em complemento, assim como a iAd, a AdMob também faz referência a uma plataforma de publicidade móvel, cuja detentora é a Google. Diferente de sua co-irmã, de exclusividade para iOS, a AdMob não fica restrita a operar em dispositivos Android, se estendendo a cenários que envolvem outros tipos de plataformas móveis, tais como WebOS, Windows Phone e Flash Lite.

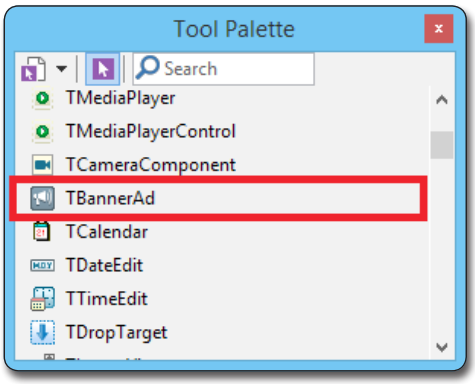


Figura 12. Grupo Additional – Novo componente TBannerAd

Em complemento a adição de novos elementos FireMonkey para o contexto Mobile, a nova versão do Delphi apresenta também atualizações importantes em componentes já existentes. Prova disso é o controle TWebBrowser (unit FMX.WebBrowser), que agora no Delphi XE6 conta com dois novos métodos: LoadFromStrings e EvaluateJavaScript. O primeiro, é responsável por exibir dentro do próprio componente WebBrowser o conteúdo string de um HTML. Para isso, opera com dois parâmetros fundamentais, onde são especificados a string HTML a ser mostrada (parâmetro Content) bem como o caminho a ser utilizado para resolver URLs relativas dentro da página carregada (parâmetro BaseUrl). Já EvaluateJavaScript possibilita a execução imediata de um código JavaScript qualquer sobre uma página HTML exibida no controle WebBrowser. Para tal, o método trabalha com um único parâmetro, do tipo string, que recebe o script customizado a ser incorporado ao contexto do HTML em questão.

FireDAC

Surgido como uma das novidades do Delphi XE3, o FireDAC é uma biblioteca de componentes de acesso a dados de terceiro que foi definitivamente incorporada ao IDE somente na versão XE5. Isto porque, até então, sua forma de distribuição aos detentores de licença da ferramenta se dava na forma de um instalador externo, tal qual um add-in. Já no Delphi XE5, dada sua crescente importância no âmbito da ferramenta, o mesmo passou a ser parte integrante da família de componentes nativos, estando naturalmente disponível na Tool Palette do IDE. Comercialmente, o FireDAC foi apresentado aos desenvolvedores como sendo uma alternativa imediata ao dbExpress (DBX), uma vez que atendia aos mesmos requisitos, porém de uma maneira mais eficaz e largamente otimizada. Em meio a protestos redigidos nos diversos fóruns da comunidade, o fato é que atualmente o FireDAC ocupa o posto de "principal opção nativa para acesso a dados" para o desenvolvimento de aplicações Delphi, anteriormente ocupada pelo dbExpress. Sendo assim, no XE6 ele se consolida como o ponto de conectividade para projetos Desktop e Mobile, operando em ambientes Windows, Mac OS X, iOS e Android.

Diante de sua reconhecida paridade com o dbExpress, a biblioteca do FireDAC acaba por apresentar componentes similares ao de seu "concorrente" nativo, o que abrange tanto a parte funcional quanto o aspecto prático de uso de cada elemento. Como exemplo, é possível citar a atuação do TClientDataSet, que complementa o uso dos componentes DBX na construção de uma aplicação de banco de dados tradicional, por meio das tratativas dos dados em memória. No FireDAC, esse tipo de gerenciamento está então relacionado ao seu componente TFDMemTable, cuja classe implementa um conjunto de dados em memória, opcionalmente relacionado a um banco de dados. Diz-se opcional pelo fato de que agora no XE6, assim como o ClientDataSet, o componente é capaz de operar com dados provindos de diferentes tipos de fontes além do banco de dados, tais como arquivos binários, XML e JSON. Em razão disso, de forma abrangente, é possível afirmar que o "TClientDataSet está para o dbExpress, assim como o TFDMemTable está para o FireDAC". Em complemento, outra novidade condizente ao componente diz respeito a sua nova capacidade em receber informações de outros elementos "DataSets", por meio de uma atribuição ainda em tempo de design (design-time), de forma totalmente RAD.

API do Windows

A Windows API, ou simplesmente WinAPI, pode ser definida como sendo um conjunto sólido de interfaces de programação, que compõem a base para tal contexto no sistema operacional da Microsoft, o Windows. Pelo fato do Delphi ter suas raízes fundamentadas nesta plataforma, seu framework contempla uma série de classes diretamente relacionadas à API do Windows, atualmente alocadas em units pertencentes ao namespace Winapi. Numa evolução deste cenário, a nova versão do Delphi conta com o acréscimo de novas funcionalidades, refletidas em novas units, voltadas a determinados aspectos do suporte à plataforma. Dentre essas novidades, ganha destaque o novo suporte à API do DirectX 11 (ver **BOX 2**), que é provido às aplicações por meio das seguintes units:

- Winapi.D3D11;
- Winapi.D3D11sdklayers;
- Winapi.D3D11Shader;
- Winapi.D3DCommon.

BOX 2. Microsoft DirectX

O Microsoft DirectX, ou apenas DirectX, é um conjunto de interfaces de programação de aplicativos (APIs – Application Programming Interface) voltada à criação de jogos e aplicativos multimídia de alto desempenho gráfico, incluindo suporte a gráficos 2-D e 3-D. De forma prática, no contexto da plataforma a qual é direcionado (Microsoft Windows), ele é o responsável por padronizar a comunicação existente entre hardware e software, fornecendo as instruções necessárias para que ambos façam o devido uso de seus recursos.

Ainda na questão de manipulação gráfica, além do DirectX, o Delphi XE6 conta também com novidades relacionadas a OpenGL (ver **BOX 3**), possibilitando a utilização dos mais recentes recursos apresentados pela API desta tecnologia. Sem deixar de mencionar, até a versão XE5, o framework do Delphi já trazia consigo uma unit específica ao contexto, denominada intuitivamente como OpenGL, também pertencente ao mesmo namespace das units relacionadas ao DirectX, o Winapi. Agora no XE6, todo este cenário é alterado,

a começar pela adição de uma nova unit ao framework de desenvolvimento, denominada Winapi.OpenGLext, e que centraliza o pleno suporte aos recursos das versões 1.2 a 4.3 do OpenGL. Em consequência, por questões de retrocompatibilidade, agora a unit Winapi.OpenGL prevê o suporte completo e exclusivo à versão 1.1 da API da tecnologia.

BOX 3. OpenGL

OpenGL é o acrônimo para o termo em inglês Open Graphics Library que, numa conjugação tradicional é utilizada para se nomear uma API de computação gráfica naturalmente voltada ao desenvolvimento de aplicativos, jogos e ambientes gráficos, incluindo 3D. Por ser uma biblioteca de caráter livre, tanto sua aceitação quanto sua utilização são bastante comuns em meio à comunidade tecnológica.

Diretórios da ferramenta

Além das novidades categorizadas a um grupo de interesse, o novo Delphi apresenta ainda outras inovações mais sutis, que podem passar despercebidas tanto aos olhos quanto na questão da usabilidade do desenvolvedor. Prova disso é que agora na versão XE6, o caminho do diretório padrão de instalação da ferramenta foi alterado. Anteriormente a este release, o próprio Delphi estabelecia um paradigma para a composição de seu diretório de instalação a partir do seguinte molde, conforme mostrado a seguir, onde X é o número de sua versão:

```
C:\Program Files (x86)\Embarcadero\RAD Studio\[X]
```

Logo, partindo deste princípio, para as instalações das versões XE4 e XE5, o caminho de ambas se dava da seguinte forma:

```
XE4
C:\Program Files (x86)\Embarcadero\RAD Studio\11.0

XE5
C:\Program Files (x86)\Embarcadero\RAD Studio\12.0
```

No XE 6 este "padrão" é alterado, sendo a corrente versão instalada no seguinte diretório:

```
C:\Program Files (x86)\Embarcadero\Studio\14.0
```

Seguindo por esta linha, o diretório de documentos públicos da ferramenta, onde são disponibilizados, por exemplo, as amostras de projetos (samples), também sofre a mesma mudança de paradigma. A fim de ilustrar este cenário, a seguir são mostrados os diretórios relacionados à versão anterior do produto (XE5) bem como a atual (XE6).

```
XE5
C:\Users\Public\Documents\Embarcadero\RADStudio\13.0\

XE6
C:\Users\Public\Documents\Embarcadero\Studio\14.0\
```

Units descontinuadas/renomeadas

A nova versão do Delphi, além de contar com o acréscimo de novas units, formaliza também a descontinuação de outras. No histórico da ferramenta, essa "descontinuidade" sempre foi refletida basicamente de três formas: pela simples exclusão da unit do framework, pela mescla de uma unit a outra existente ou ainda pela substituição de uma unit, por outra nova. No caso do Delphi XE6, se aplicam as duas últimas opções. Pelo fato do framework da ferramenta atualmente contar com duas vertentes – VCL e FMX – sendo a primeira, de longe, a mais estável, todas estas mudanças aqui mencionadas se referem ao contexto do FireMonkey. Logo, as units que não estão mais presentes na estrutura da FMX nesta versão do IDE são listadas a seguir:

- FMX.Sensors (agora inclusa em System.Messaging);
- FMX.PixelFormats (agora FMX.Types);
- FMX.Messages (agora inclusa em System.Sensors).

Adicionalmente a este panorama, a nova versão da ferramenta apresenta também algumas unidades de código que foram renomeadas, seja por questões funcionais ou de organização. Como exemplo maior está a unit FireDAC.Phys.DS, presente no XE5, e agora apresentada como FireDAC.Phys.DataSnap no XE6.

Tethering Application e BaaS

De forma proposital, duas das principais novidades trazidas pelo Delphi XE6 foram deixadas para este final, dada a extensibilidade do assunto relacionado a cada uma delas. Logo, para o devido entendimento conceitual e prático, seria necessário um artigo exclusivo para cada tema. Todavia, a introdução dada aqui a cada uma dessas novidades servirá, de forma natural, como ponto de partida a um eventual aprofundamento posterior. No XE6, as novidades citadas se refletem em dois novos termos – Tethering e BaaS – que são disponibilizados no IDE na forma de dois novos grupos de componentes (Tethering e BAAS Client) na Tool Palette, conforme mostrado na **Figura 13** e **14**.

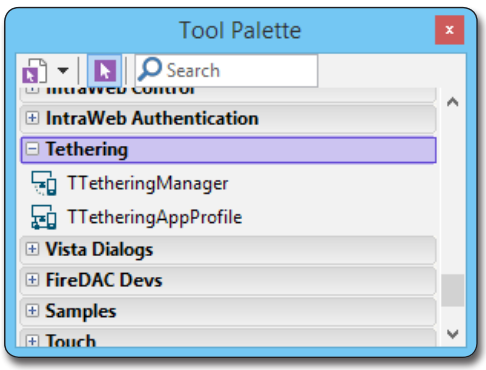


Figura 13. Novo grupo de componentes - Tethering

Tethering

Em suma, o termo Tethering remete a um novo recurso provido pelo Delphi responsável pela interação entre aplicações de diferentes tipos, tal como a conexão de uma aplicação VCL para Desktop

a uma aplicação FMX para Mobile. Assim, na prática, através do uso de seus componentes relacionados torna-se possível, por exemplo, a extensão de uma aplicação Windows existente para uma plataforma Mobile (iOS e Android), com poucos ajustes, sem a necessidade de se portar essencialmente toda a aplicação. Por abranger cenários distintos de desenvolvimento (VCL e FMX), toda a infra do recurso de Tethering se concentra na RTL (Run-Time Library) do Delphi.

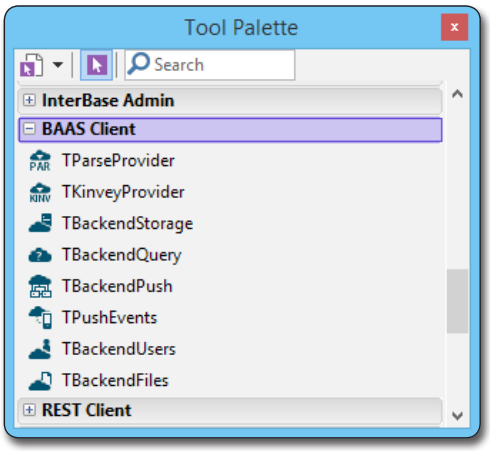


Figura 14. Novo grupo de componentes – BAAS Client

BaaS

O termo "Backend as a Service", ou simplesmente BaaS, refere-se a um novo framework REST (REST BaaS framework) suportado pelo Delphi, que permite que uma aplicação móvel seja integrada a provedores de serviços na nuvem (Cloud), líderes de mercado nesta segmentação, tais como os providers Kinvey e Parse. Assim, retira-se do desenvolvedor toda a responsabilidade por construir e manter serviços de Backend para serem consumidos por suas aplicações, tirando proveito de soluções prontas e ideais. Em linhas gerais, na prática, o uso dos componentes do grupo BAAS Client possibilita a essas aplicações:

- Armazenamento de objetos na nuvem;
- Manipular objetos (criação, recuperação, atualização e etc.);
- Efetivar tratativas de usuários (autenticação, criação, atualização e etc.);
- Consultar elementos envolvidos;
- Enviar notificações "Push".

Com esta versão a Embarcadero firma sua estratégia de oferecer uma ferramenta sempre atualizada, que produz aplicações nativas para Windows, Mac, iOS e Android. Trazendo novidades para os principais pontos que caracterizam essa realidade do Delphi, temos uma ferramenta que atende as mais variadas solicitações dos desenvolvedores.

Autor

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ e pós-graduado em Engenharia de Componentes. Atua como Analista Programador Delphi há 9 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor. Site pessoal: www.fabriciokawata.com.br

Links:

RAD Studio XE6 – Página oficial do produto
http://www.embarcadero.com/products/rad-studio

RAD Studio XE6 – Download Trial
https://downloads.embarcadero.com/free/rad_studio

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

SEU CARRO TEM SEGURO, SUA SAÚDE TEM SEGURO, MAS E O SEU EMPREGO... TÁ SEGURO??

NÃO DEIXE JUSTAMENTE A SUA CARREIRA FICAR EM RISCO!

Manter-se atualizado com todas as novidades do mercado de desenvolvimento é obrigação de todo bom programador. Faça agora mesmo um seguro para a sua carreira. Seja um assinante MVP!

Saia do risco!

TENHA ACESSO A:

- +DE **260** CURSOS ONLINE
- **09** REVISTAS MENSAIS
- **7.850** VÍDEO-AULAS

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

QUEM TEM ESTÁ TRANQUILO.

POR APENAS **69**,90 MENSAIS

DEVMEDIA | Acesse: www.devmedia.com.br/mvp

SVN e Jenkins: Integração Contínua

Integração contínua é uma prática de metodologia ágil diretamente ligada ao XP (eXtreme Programming). A integração continua visa garantir que, qualquer alteração no código fonte seja rapidamente integrada e validada dentro da solução completa de forma transparente e automatizada. Caso uma alteração que tenha sido integrada acabe por gerar uma falha, o processo de integração continua deve garantir que o feedback seja imediato e preciso.

Para validar o resultado de um processo de integração contínua é possível mesclar inúmeras etapas. Diferente do que muitos pensam, integração continua não se resume somente ao build automatizado, mas sim, em uma série de etapas, com destaque para os testes automatizados.

Resumindo, segundo Martin Fowler: "Integração Contínua é uma prática de desenvolvimento de software onde os membros de um time integram seu trabalho frequentemente, geralmente cada pessoa integra pelo menos diariamente – podendo haver múltiplas integrações por dia. Cada integração é verificada por um build automatizado (incluindo testes) para detectar erros de integração o mais rápido possível. Muitos times acham que essa abordagem leva a uma significante redução nos problemas de integração e permite que um time desenvolva software coeso mais rapidamente."

A **Figura 1** ilustra um exemplo de ciclo de integração contínua.

Os benefícios na adoção de um processo de integração contínua são inúmeros, porém, podemos destacar alguns imediatos:

- Agilidade na detecção de falhas;
- Build automatizado e parametrizado;
- Atualização de ambiente de testes automatizado;
- Aumento da confiabilidade dos artefatos produzidos;
- Processo de desenvolvimento em equipe mais seguro.

Veremos agora um pouco sobre cada ferramenta que será utilizada no artigo e quais são os seus propósitos.

Jenkins

O Jenkins é uma ferramenta multiplataforma Open Source voltada para a integração contínua, baseada

Fique por Dentro

O artigo apresenta a prática de integração contínua, muito utilizada em ambientes que adotam o desenvolvimento ágil, porém, não é muito conhecido em meio a comunidade Delphi. Sua aplicação objetiva a garantia de uma entrega de software melhor, com qualidade, através de rotinas automatizadas que vão desde compilação aos testes. Para demonstrar isso o artigo aborda a criação do ambiente ideal utilizando ferramentas gratuitas, como o Jenkins para a realização de Builds, o VisualSVN para controle de versões e DUnit para testes automatizados, tudo isso com um exemplo prático.

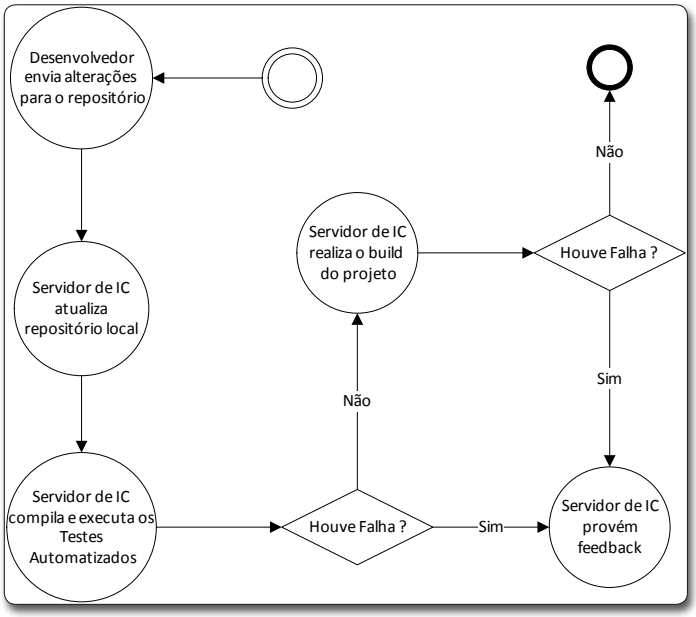


Figura 1. Exemplo de ciclo de integração contínua (I.C.)

(Fork – ver **BOX 1**) no projeto Hudson. Basicamente o Jenkins funciona como um servidor de integração continua onde é possível programar Jobs para diversos projetos, inclusive sendo possível criar integrações entre esses Jobs. Não entraremos em muitos detalhes sobre o Jenkins por não ser o foco deste artigo, mas é altamente recomendado visitar o site oficial (seção **Links**) e se associar ao grupo brasileiro sobre a ferramenta (jenkinsci-br @googlegroups.com).

Uma das vantagens do Jenkins é a sua comunidade extremamente forte.

Neste artigo iremos utilizar a sua distribuição para o ambiente Windows visando uma melhor integração com o Delphi.

BOX 1. Fork

O Fork, recurso do GitHub, consiste em criar uma cópia independente de um determinado repositório, desta forma, é possível trabalhar em um fork de um projeto, sem que as alterações afetem o projeto original. Esta é a forma utilizada para contribuir em projetos open source. Após as alterações serem finalizadas no fork, é possível realizar um pull request no repositório original, com isso o dono do projeto pode tomar a decisão de acoplar ou não as alterações do fork ao projeto original.

Subversion (SVN)

O Subversion é um dos sistemas controladores de versão mais famosos atualmente. Utilizaremos ele como nosso repositório neste artigo por ser extremamente simples e integrar perfeitamente com o Delphi e com o Jenkins (outra vantagem do Jenkins é sua vasta gama de Plugins que permite a integração com os mais diversos sistemas controladores de versão como por exemplo o GIT).

Iremos criar um servidor Subversion Windows e nele iremos criar nosso repositório, para isso, utilizaremos o **VisualSVN Server** que pode ser obtido no site disponível na seção **Links**.

MSBuild

O MSBuild é a plataforma de compilação da Microsoft que pode ser utilizada em conjunto com o compilador do Delphi. Utilizaremos um Plugin para o Jenkins que se integra com o MSBuild afim de realizar os processos de compilação dos nossos Jobs de projetos Delphi.

O MSBuild já é disponibilizado no Windows e pode ser encontrado no diretório C:\Windows\Microsoft.NET\Framework\[VERSAO]\MSBuild. Neste artigo utilizaremos a versão v3.5.

Instalando o SVN e criando um repositório

Finalmente iremos colocar a mão na massa, começando por nosso repositório no SVN. Para isso, iremos instalar o VisualSVN Server, uma versão do SVN para Windows.

Basicamente seu processo de instalação se resume ao padrão Windows (next, next e finish). A única observação se dá na escolha da edição (VisualSVN Server Editions), como pode ser visto na **Figura 2**. Nesta tela selecione a opção Standard Edition que é a versão gratuita.

Após a instalação, será apresentada uma tela como a da **Figura 3**, onde será possível criar nossos usuários e nossos repositórios.

Criaremos dois usuários, um para o desenvolvedor e outro para o Jenkins, para isso, clique com o botão direito do mouse em Users e depois clique em Create User. Após preencher o nome de usuário e senha, o usuário já está criado, agora basta criarmos o repositório. Para isso, clique com o botão direito do mouse em Repositories e depois clique em Create New Repository.

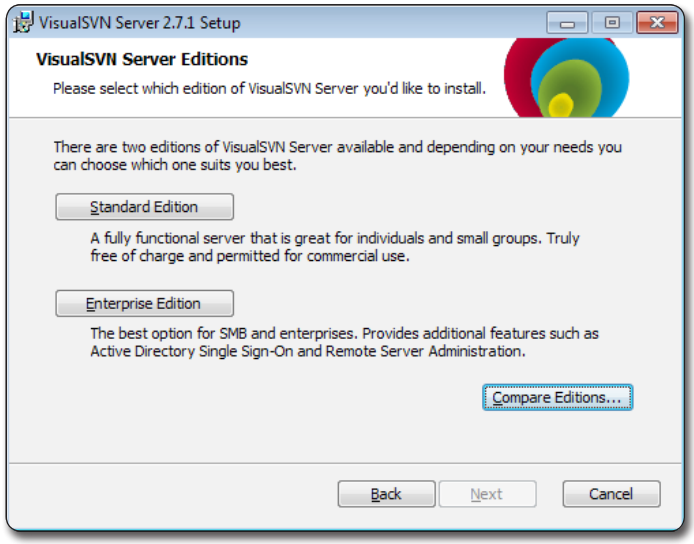


Figura 2. Instalação do VirtualSVN Server

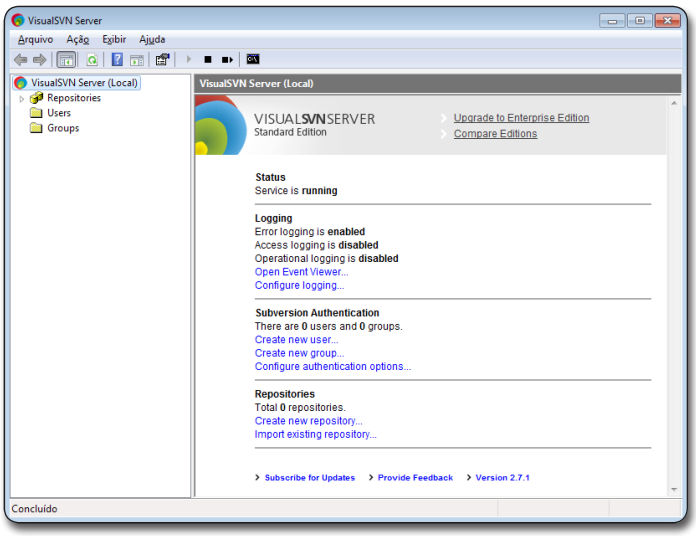


Figura 3. VisualSVN Server

Após preencher seu nome, serão apresentadas as opções de estrutura inicial do repositório, selecione a opção Single-Project Repository para que já seja criada uma estrutura padrão (Branches, Trunk e Tags). Após isso, informe que todos os usuários têm acesso completo ao repositório e o processo estará finalizado (**Figura 4**).

Fazendo o primeiro Checkout

Para se trabalhar com o SVN, é preciso fazer um checkout do repositório em sua máquina, ou seja, fazer uma cópia de trabalho do repositório. Para isso utilizaremos o próprio Delphi que já possui integração nativa com o SVN.

No Delphi, em File->Open From Version Control, coloque a url do repositório e o diretório da cópia de trabalho para realizar o checkout, conforme é demonstrado na **Figura 5**.

Pronto, agora basta criar o projeto e salvar no diretório da sua cópia local. Vale uma observação, o SVN trabalha com o conceito

de Branches para a cópia de trabalho, Trunk para a última versão estável e Tags para as versões antigas.

Criando o projeto em Delphi

Iremos criar um projeto em Delphi simples que se resume a uma classe somente que será utilizada para validar nossos conceitos. Esta classe deverá realizar cálculos relacionados a desconto e posteriormente, iremos criar uma rotina de automatização de testes para esta. Crie um projeto do tipo VCL Forms Application ou FireMonkey HD Application e dê o nome de ClubeDelphi. Inclua uma unit chamada untDesconto e nesta codifique a classe TDesconto, conforme a **Listagem 1**.

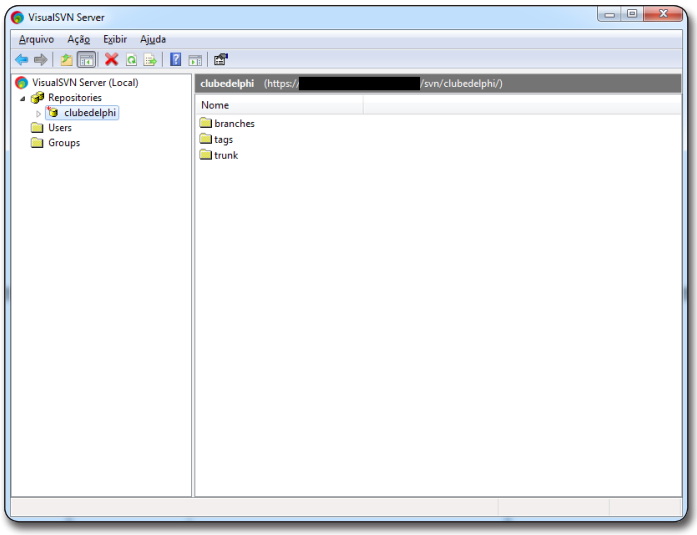


Figura 4. Repositório Criado

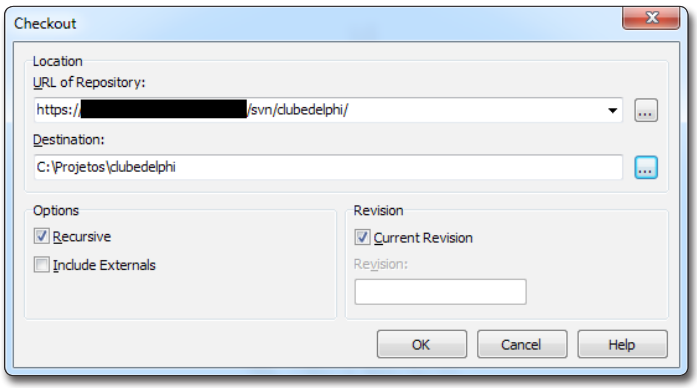


Figura 5. Realizando Checkout

Como podemos ver a classe TDesconto, basicamente calcula descontos por porcentagem e por valor. Antes de prosseguirmos com o projeto de testes automatizados, iremos incluir nosso projeto ClubeDelphi em nosso repositório do SVN, para isso, salve o projeto no diretório onde foi feito o checkout do nosso repositório do SVN (**Figura 5**). Após isso, clique com o botão direito do mouse no projeto em Project Manager e depois em Add to Version Control, conforme a **Figura 6**.

Listagem 1. Classe que realiza cálculos de descontos

```
01 unit untDesconto;

02 interface

03 type
04  TDesconto = class
05  strict private
06    FValorTotal: Real;
07    FValorFinal: Real;
08    FDescontoValor: Real;
09    FDescontoPorcentagem: Real;
10    procedure SetValorTotal(const Value: Real);
11  public
12    function AplicarDescontoPorcentagem(const APorcentagem : Real):Real;
13    function AplicarDescontoValor(const AValor : Real):Real;
14    property ValorTotal : Real read FValorTotal write SetValorTotal;
15    property DescontoPorcentagem : Real read FDescontoPorcentagem;
16    property DescontoValor : Real read FDescontoValor;
17    property ValorFinal : Real read FValorFinal;
18  end;

19 implementation

20 function TDesconto.AplicarDescontoPorcentagem(
     const APorcentagem: Real): Real;
21 begin
22   Self.FDescontoPorcentagem := APorcentagem;
23   Self.FDescontoValor      := (APorcentagem * Self.FValorTotal) / 100;
24   Self.FValorFinal         := Self.FValorTotal - Self.FDescontoValor;

25   Result := Self.FValorFinal;
26 end;

27 function TDesconto.AplicarDescontoValor(const AValor: Real): Real;
28 begin
29   Self.FDescontoValor      := AValor;
30   Self.FDescontoPorcentagem := (AValor * 100) / Self.FValorTotal;
31   Self.FValorFinal         := Self.FValorTotal - Self.FDescontoValor;

32   Result := Self.FValorFinal;
33 end;

34 procedure TDesconto.SetValorTotal(const Value: Real);
35 begin
36   FValorTotal := Value;
37   Self.AplicarDescontoValor(Self.FDescontoValor);
38 end;

39 end.
```

Feito isso, nosso projeto já faz parte do repositório. Agora iremos criar o projeto de testes.

Criando o projeto de testes automatizados

Um processo de integração contínua não faz sentido se não possuir uma rotina de testes automatizados, pois só o fato de realizar um Build em um projeto, não garante que ele está livre de falhas. Para Delphi existe o projeto Open Source DUnit que já vem acoplado a IDE desde algumas versões mais antigas. Para mais informações sobre, veja a seção **Links** ao final do artigo.

Criar um projeto de testes é algo muito simples, basta acessar o menu em File>New>Other>Unit Test>Test Project. Será apresentada uma tela semelhante à **Figura 7**.

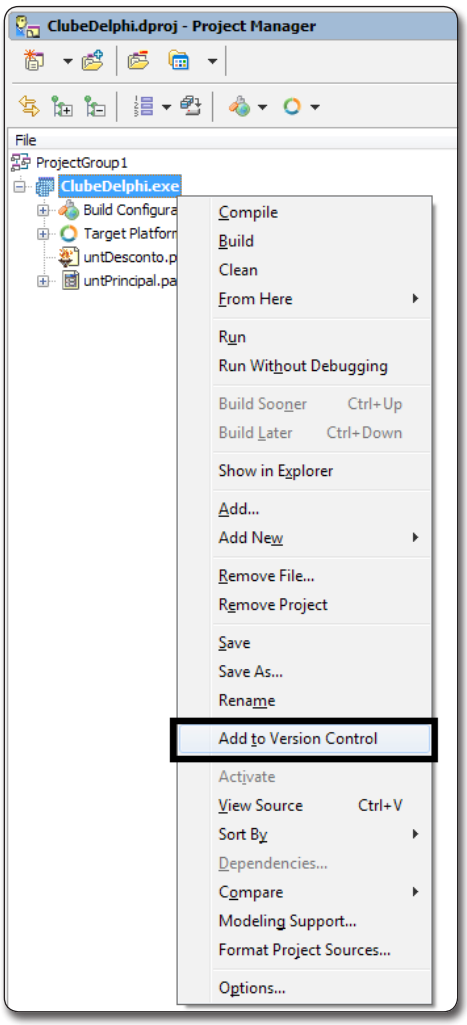


Figura 6. Adicionando arquivos ao repositório

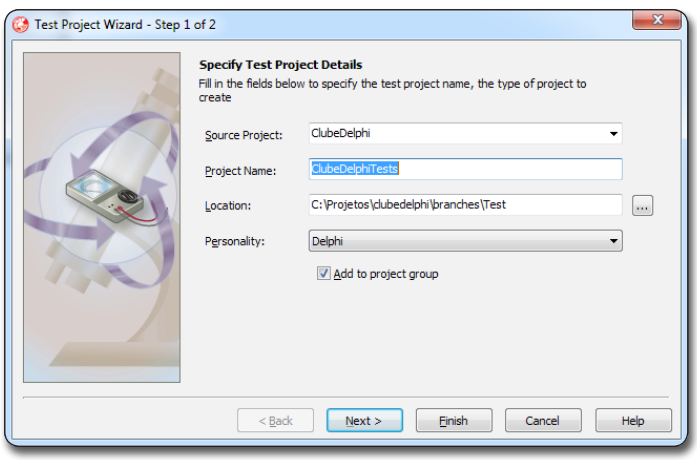


Figura 7. Criando um projeto de testes automatizados

Basta informar o projeto principal (Source Project), o nome do projeto de testes automatizados (Project Name) e a localização dos arquivos do projeto de testes automatizados (Location). Assim, o projeto de testes automatizados está criado. Faremos então o nosso primeiro caso de testes, baseado na classe TDesconto

Novamente acesse o menu File>New>Other>Unit Test>Test Case. Será apresentada uma tela semelhante à **Figura 8**.

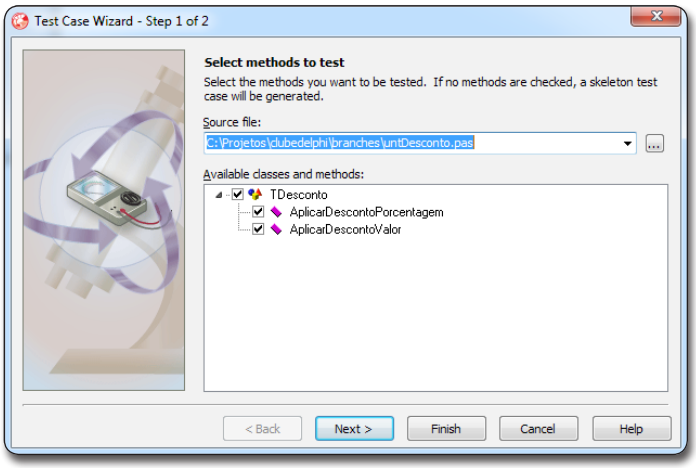


Figura 8. Criação do caso de testes

Estes Wizards facilitam muito a vida de quem está começando a lidar com a DUnit, porém, nada impede que você crie seus testse do zero, basta conhecer o framework da DUnit. Após a criação do caso de testes, teremos uma unit com o nome de TestUntDesconto contendo a classe TestTDesconto com o esqueleto de uma rotina de automatização de testes baseada na classe TDesconto. Após algumas alterações na classe TestTDesconto, esta deve ficar semelhante à **Listagem 2**.

O objetivo desse código não é abordar em si o que a unit TestDesconto faz, mas sim o uso do DUnit. Para isso, vale conhecermos a respeito de alguns conceitos básicos. As classes de caso de teste devem descender da classe TTestCase para que seja possível registrar a suíte de testes da classe. Os métodos SetUp (linhas 17 a 21) e TearDown (linhas 22 a 26) podem ser sobrepostos (override) caso sejam necessários. Esses são executados sempre antes (SetUp) e depois (TearDown) da execução de qualquer método de teste. Metodologias ágeis defendem que cada teste deve ser independente, por isso, é possível criar e liberar a instancia de um objeto a cada teste, prevenindo assim qualquer possível interferência entre os testes, para isso, basta usar o método SetUp e o método TearDown. Os métodos de teste devem estar na sessão **published** da classe de caso de testes, para que assim, através da RTTI o framework da DUnit consiga executá-los.

Voltando a falar da nossa classe TestTDesconto, ela basicamente testa as 3 possibilidades da classe TDesconto, aplicação de desconto por porcentagem (linhas 27 a 38), desconto por valor (linhas 39 a 50) e atualização do valor total com recálculo dos descontos (linhas 51 a 59). Para ver nossos testes em ação, basta executar o aplicativo. Essa ação deverá apresentar algo como a **Figura 9**.

Mais sobre o SVN no Delphi

Finalizado o projeto de testes, é chegada a hora de adicioná-lo ao nosso repositório no SVN, para isso, repita o processo de Add to Version Control, como demonstrado na **Figura 6**.

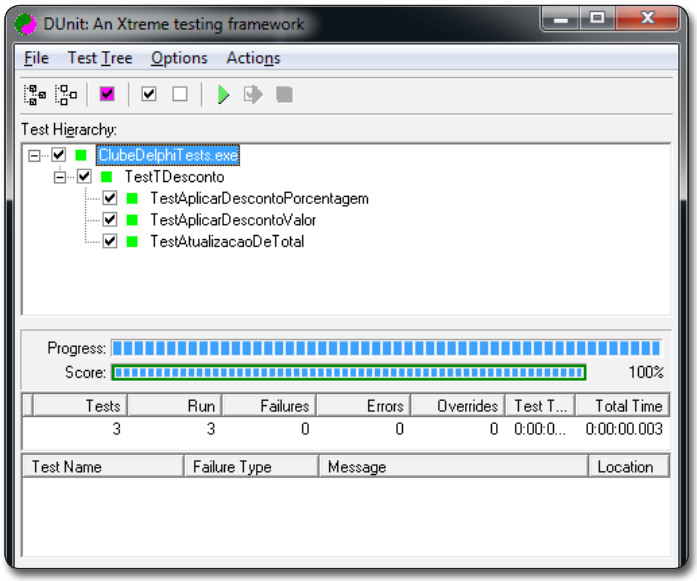


Figura 9. DUnit em ação

Feito isso, nosso repositório já sabe da existência dos arquivos do nosso projeto, agora precisamos realizar o commit das alterações. O commit é o processo onde as alterações locais são efetivadas no repositório principal.

Para realizar o commit a partir do Delphi, basta clicar com o botão direito do mouse no projeto em Project Manager e depois clicar em Subversion>Commit>From Project Directory. Será apresentada uma tela semelhante à **Figura 10**.

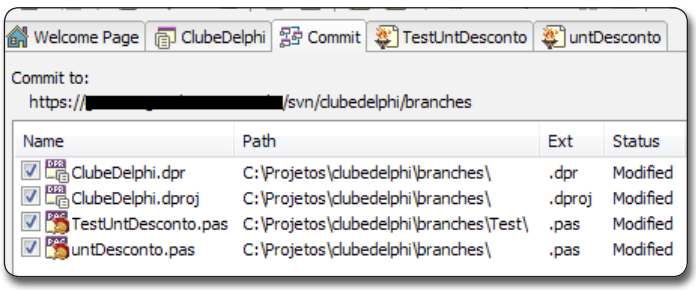


Figura 10. Arquivos que foram alterados após o último commit

Listagem 2. Classe que realiza cálculos de descontos

```
01 unit TestUntDesconto;

02 interface

03 uses TestFramework, untDesconto;

04 type

05  TestTDesconto = class(TTestCase)
06  strict private
07    FDesconto: TDesconto;
08  public
09    procedure SetUp; override;
10    procedure TearDown; override;
11  published
12    procedure TestAplicarDescontoPorcentagem;
13    procedure TestAplicarDescontoValor;
14    procedure TestAtualizacaoDeTotal;
15  end;

16 implementation

17 procedure TestTDesconto.SetUp;
18 begin
19  FDesconto := TDesconto.Create;
20  FDesconto.ValorTotal := 250;
21 end;

22 procedure TestTDesconto.TearDown;
23 begin
24  FDesconto.Free;
25  FDesconto := nil;
26 end;

27 procedure TestTDesconto.TestAplicarDescontoPorcentagem;
28 var
29  ReturnValue: Real;
30  APorcentagem: Real;
31 begin
32  APorcentagem := 10;
33  ReturnValue := FDesconto.AplicarDescontoPorcentagem(APorcentagem);
34  CheckEquals(225,ReturnValue,'Retorno do método');
35  CheckEquals(225,FDesconto.ValorFinal,'Valor final');
36  CheckEquals(25,FDesconto.DescontoValor,'Desconto em valor');
37  CheckEquals(10,FDesconto.DescontoPorcentagem,'Desconto em
    porcentagem');
38 end;

39 procedure TestTDesconto.TestAplicarDescontoValor;
40 var
41  ReturnValue: Real;
42  AValor: Real;
43 begin
44  AValor     := 25;
45  ReturnValue := FDesconto.AplicarDescontoValor(AValor);
46  CheckEquals(225,ReturnValue,'Retorno do método');
47  CheckEquals(225,FDesconto.ValorFinal,'Valor final');
48  CheckEquals(25,FDesconto.DescontoValor,'Desconto em valor');
49  CheckEquals(10,FDesconto.DescontoPorcentagem,'Desconto em
    porcentagem');
50 end;

51 procedure TestTDesconto.TestAtualizacaoDeTotal;
52 begin
53  FDesconto.AplicarDescontoValor(35);
54  CheckEquals(215,FDesconto.ValorFinal);
55  FDesconto.ValorTotal := 350;
56  CheckEquals(35,FDesconto.DescontoValor,'Desconto $ pós atualização');
57  CheckEquals(10,FDesconto.DescontoPorcentagem,'Desconto % pós
    atualização');
58  CheckEquals(315,FDesconto.ValorFinal,'Valor final pós atualização');
59 end;

60 initialization
61  RegisterTest(TestTDesconto.Suite);

62 end.
```

Nos casos em que um membro da equipe realizar um Commit no repositório e sua cópia passar a ficar desatualizada, basta realizar o processo de Update. O Update é o processo oposto ao Commit, se no Commit estamos efetivando nossas alterações locais no repositório principal, no Update estamos atualizando nossa cópia local se baseando na cópia do repositório. No Delphi esta opção fica em Subversion>Update>From Project Directory, ou ao clicar de direito sobre o projeto no Project Manager.

Jenkins

Vamos agora para o nosso "maestro", que irá orquestrar toda a nossa obra, o poderoso servidor de integração contínua Jenkins.

Lembrando que o Jenkins não é a única opção de servidor de integração continua no mercado, como é o caso do Final Builder, um excelente servidor de integração que já vem (em uma versão um pouco mais limitada) junto a algumas versões do Delphi.

A instalação do Jenkins não possui relativas dificuldades. Depois de instalado, o Jenkins passa a rodar como um serviço do Windows e responde com um servidor HTTP na porta 8080 (**Figura 11**).

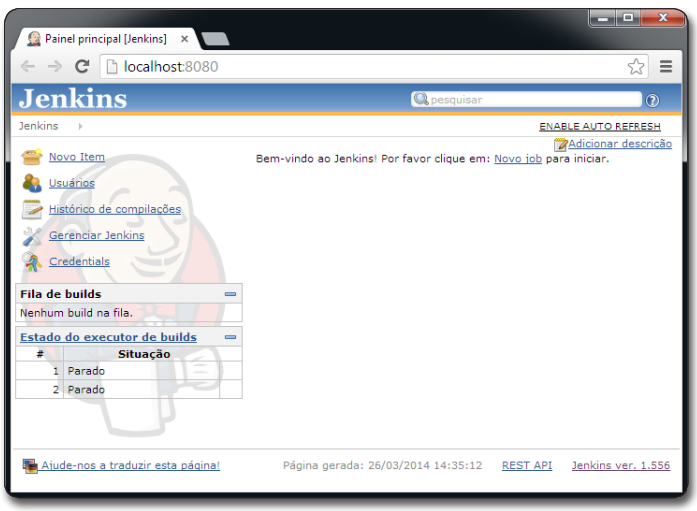


Figura 11. Tela inicial do Jenkins

Com o Jenkins instalado e funcionando, podemos instalar o Plugin do MSBuild para que seja possível compilar nossas aplicações Delphi. Na lateral esquerda da tela principal (**Figura 11**) clique sobre o menu Gerenciar Jenkins. Este é o painel de controle do Jenkins. Agora vá em "Gerenciar Plugins>Disponíveis" e na caixa de filtro, digite MSBuild. Marque para instalar o Plugin e clique em "Baixar agora, instalar e depois reiniciar".

Feito isso, o MSBuild já estará instalado e, caso haja algum problema na instalação automática do plugin, é possível realizar o download manualmente (ver seção **Links**) e realizar a instalação em "Gerenciar Plugins>Avançado>Upload". Porém, se você usar proxy na sua rede, pode ser que ocorra algum erro para realizar o download, caso você se depare com esse cenário, vá em "Gerenciar Plugins> Avançado" e realize a configuração de proxy do Jenkins.

Outro Plugin necessário é o EnvInject (seção **Links**) que basicamente injeta variáveis de ambiente ao processo de build de um projeto. Instale-o da mesma forma como foi instalado o MSBuild.

Configurando o MSBuild

Com nossos Plugins instalados no Jenkins, vamos realizar as configurações do MSBuild na opção "Gerenciar Jenkins>Configurar o sistema>MSBuild Instalações". Nesta sessão, iremos especificar as informações sobre o MSBuild instalado no servidor que iremos utilizar. Os parâmetros que iremos configurar são os seguintes:

- **Name:** Um nome de identificação para o Plugin instalado, já que podem ser configurados mais de um MSBuild.
- **Path do MSBuild:** Caminho completo do binário do MSBuild, geralmente encontrado em "C:\Windows\Microsoft.NET\Framework\[versão]\MSBuild.exe".

Criando um Job

Primeiramente iremos criar um Job simples que somente baixe os fontes do SVN e compile o projeto, depois iremos aprimorando. Clique em "Novo Item" na lateral esquerda do Jenkins, informe o nome do Job e selecione a opção "Construir um projeto de software free-style".

Na tela de parametrização do Job, na opção "Gerenciamento de código fonte", selecione o Subversion. Ao realizar essa seleção, serão apresentadas as parametrizações do gerenciador. Para o Subversion, iremos configurar somente o diretório do nosso repositório, algo como https://[servidor]/svn/clubedelphi/branches/. Caso o Jenkins não consiga estabelecer conexão com o repositório, será solicitado o usuário e senha de acesso através do link "Enter credential". Especifique o usuário criado no SVN para ser usado no Jenkins com o tipo de autenticação "username/password".

Agora na opção "Ambiente de Build", marque a opção "Inject environment variable to the build process" (essa opção está disponível graças ao Plugin EnvInject) e em "Properties Content" copie o conteúdo do arquivo rsvars.bat sem os @SET antes do nome das variáveis. Esse arquivo já vem com o Delphi, geralmente no diretório "C:\Program Files (x86)\Embarcadero\RAD Studio\[versão]\bin". Essas variáveis de ambiente são necessárias no momento do Build de aplicações Delphi pelo MSBuild.

Finalmente chega a hora de configurar os processos de Build. Na tela de parametrização do Job, vá em "Adicionar passo no build>Build a Visual Studio project or solution using MSBuild". Em "MSBuild Version", selecione a instalação do MSBuild configurada anteriormente, em "MSBuild Build File", digite o nome do arquivo dproj do projeto que será compilado, no caso ClubeDelphi.dproj e por fim, em "Command Line Arguments", coloque o seguinte texto: "/v:d /target:Build /p:config=Debug", com isso estamos especificando que faremos um build do projeto no modo Debug.

Com essas configurações feitas, deveremos ter algo como a **Figura 12**.

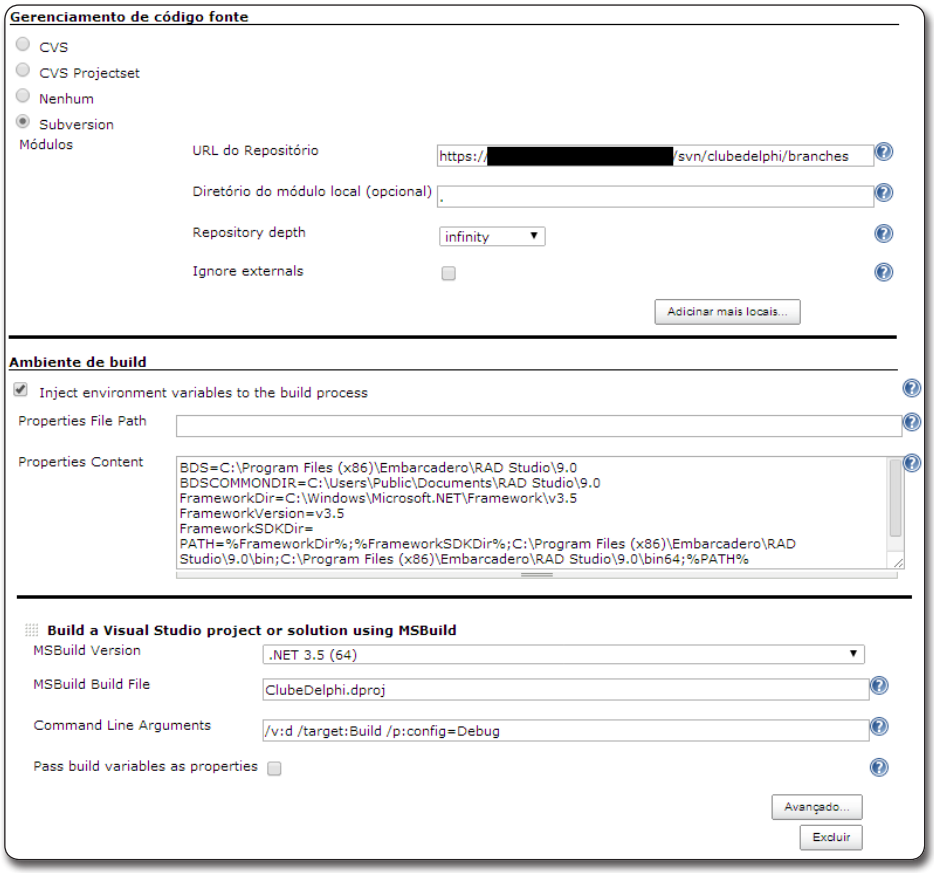


Figura 12. Parametrização do Job

Salve as configurações e clique em "Construir agora" na lateral esquerda, para testar o Job. Se tudo ocorrer como esperado, deverá aparecer um círculo azul no histórico de Builds dentro das opções do Job, ou no grid da Dashboard do Jenkins, isso significa que o processo foi bem sucedido, caso algo dê errado, esse círculo deverá ser vermelho. Observe o exemplo na **Figura 13**.

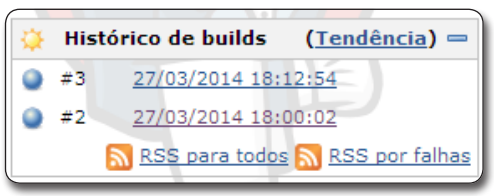


Figura 13. Histórico de Builds

Executando Testes pelo Jenkins

Um processo de integração contínua não está completo se não existirem testes unitários automatizados. Sendo assim, nosso projeto de testes já está pronto, porém, da forma como o Delphi cria o projeto de teste da DUnit através de seu Wizard, o mesmo não está pronto para a execução por linha de comando. A **Listagem 3** apresenta o código do arquivo .dpr recém gerado de um projeto de teste da DUnit.

Porém, precisamos que o aplicativo de testes funcione com interface gráfica se for executado por uma pessoa, e também como console, se for executado pelo Jenkins. Assim sendo, faremos algumas alterações neste arquivo para que o mesmo suporte as duas opções. O código deve ser feito como descrito na **Listagem 4**.

Embora esse código seja relativamente grande, é relativamente simples. Devemos atentar principalmente a partir da linha 15. Nota-se que é verificado se foi recebido um parâmetro denominado 'jenkins'. Caso ele exista, a execução ocorrerá da linha 17 a 25, onde é executado o método RunRegistered-Tests de TextTestRunner (linha 17) em modo console, caso contrário, o mesmo método é executado, porém de GUITestRunner (linha 28), utilizando interface gráfica.

Voltando ao Jenkins, faremos agora com que nosso Job saiba da existência do projeto de testes e passe a considerá-lo, mas, antes disso, deve-se realizar o commit das alterações do projeto para o SVN.

Nas configurações do Job inclua mais dois passos de Build, o primeiro será um "Build a Visual Studio project or solution using MSBuild", como feito anteriormente, mas dessa vez, apontando para o projeto de testes, ou seja, no campo MSBuild File deverá estar o valor "Test\ClubeDelphiTests.dproj". Posicione esse Build como sendo o primeiro da lista.

O segundo passo de Build a ser adicionado é do tipo "Executar no comando do Windows". Esse será o passo onde iremos executar nosso teste. No campo para especificar o que será executado, informe a instrução: "Test\Win32\Debug\ClubeDelphiTests.exe jenkins". Ou seja, através de linhas de comando estaremos executando a aplicação de testes passando o parâmetro 'jenkins'. Posicione esse passo de build como sendo o segundo.

Agora o build do Job deverá conter os seguintes passos: "Compilação do projeto de testes>Execução dos testes>Compilação do projeto principal". Quando qualquer um destes passos falharem, o seguinte não será executado.

Salve as alterações e execute o Job clicando em "Construir agora". Neste momento, perceba que nada de diferente aconteceu, isso porque todos os testes estão obtendo sucesso. Para validar o ciclo, na unit "TestUntDesconto.pas" do projeto de testes, localize o método "TestTDesconto.TestAplicarDescontoPorcentagem" e altere a linha:

```
APorcentagem := 10;
```

para

```
APorcentagem := 11;
```

Listagem 3. Projeto de testes

```
01 program ClubeDelphiTests;

02 {$IFDEF CONSOLE_TESTRUNNER}
03 {$APPTYPE CONSOLE}
04 {$ENDIF}

05 uses
06   DUnitTestRunner,
07   TestUntDesconto in 'TestUntDesconto.pas',
08   untDesconto in '..\untDesconto.pas';

09 {$R *.RES}

10 begin
11   DUnitTestRunner.RunRegisteredTests;
12 end.
```

Listagem 4. Projeto de testes

```
01 program ClubeDelphiTests;

02 {$IFDEF CONSOLE_TESTRUNNER}
03 {$APPTYPE CONSOLE}
04 {$ENDIF}

05 uses
06   TestFrameWork,
07   TextTestRunner,
08   GUITestRunner,
09   TestUntDesconto in 'TestUntDesconto.pas',
10   untDesconto in '..\untDesconto.pas';

11 {$R *.RES}

12 var
13   Results: TTestResult;

14 begin
15   if ParamStr(1) = 'jenkins' then
16   begin
17     Results := TextTestRunner.RunRegisteredTests;
18     try
19       if Results.FailureCount + Results.ErrorCount = 0 then
20         ExitCode := 0
21       else
22         ExitCode := 1;
23     finally
24       Results.Free;
25     end;
26   end
27   else
28     GUITestRunner.RunRegisteredTests;
29 end.
```

Faça o commit das alterações e execute o Job. Sim, agora temos um círculo vermelho. Isso significa que algo falhou (mas para nós significa que o processo está coerente).

Para saber mais sobre a execução do Build, clique no link da execução do Build (o link que possui o círculo com a cor de status, o número de Build, e a data e hora de execução) e depois clique em Saída do Console. Para o Job, a saída do Build provavelmente ocorrerá na seguinte ordem: execução do Plugin EnvInject, atualização do repositório do SVN, compilação do projeto de testes e finalmente, a execução dos testes com o texto da **Listagem 5**.

Listagem 5. Resultado dos testes

```
DUnit / Testing
...F..
Time: 0:00:00.0

FAILURES!!!
Test Results:
Run: 3
Failures: 1
Errors: 0
There was 1 failure:
  1) TestAplicarDescontoPorcentagem: ETestFailure
    at "Retorno do método, expected: <225> but was: <222,5>"

C:\Program Files (x86)\Jenkins\jobs\ClubeDelphi\workspace>exit 1
Build step 'Executar no comando do Windows' marked build as failure
Finished: FAILURE
```

Com isso comprovamos que nosso ciclo de integração continua está funcionando por completo. Porém, há como aprimorar ainda mais a saída de console dos testes, para isso, é possível utilizar o Plugin NUnit para o Jenkins, que é específico para interagir com os resultados de testes no padrão NUnit. Instale esse plugin através da listagem de plugins disponíveis do jenkins, caso ele não esteja disponível nesta listagem, você pode baixar a sua última versão pelo GitHub (seção **Links**). A DUnit que vem com o Delphi já está preparada para exportar os resultados dos testes registrados para um arquivo XML, porém, esse arquivo não está no padrão esperado pelo plugin NUnit do Jenkins. Para resolver este problema foi criada por Mark Pickersgill a unit "XMLTestRunner2.pas" que basicamente substitui a unit "TextTestRunner.pas" em nosso exemplo. Baixe essa unit (seção **Links**) e adicione a biblioteca da DUnit do Delphi (geralmente em C:\Program Files (x86)\Embarcadero\RAD Studio\[versão]\source\DUnit\src).

Com todas ferramentas preparadas para interagir com o padrão NUnit nos resultados dos testes, é necessário adaptar a aplicação e o Job para tirar proveito deste recurso. Começaremos pelo projeto de testes. Basicamente a alteração necessária será substituir a unit TextTestRunner pela XMLTestRunner2 em nosso arquivo DPR e depois especificar o local de saída para o arquivo XML gerado. Após a alteração, o código do projeto de testes deve ficar semelhante ao da **Listagem 6**.

Note que o código é o mesmo da anterior, diferindo apenas na linha 17, onde é utilizada a unit XMLTestRunner2.

Faça o commit das alterações para o SVN. Por fim, iremos adaptar nosso Job para ter conhecimento do arquivo DUnitResult.xml que será gerado a cada execução dos testes. Nas configurações do Job, adicione um passo pós-build em "Add post-build action", esse passo será do tipo "Publish NUnit test result report" (essa opção está disponível graças à instalação do plugin NUnit). Note que a única configuração do passo é o caminho do arquivo XML. Informe o nome que especificado no projeto de testes, ou seja, "DUnitResult.xml". Salve as configurações e execute o Job.

Note que agora existe uma série de indicadores de testes, baseados no resultado do último Build. Na página principal do

Job agora é possível ver o último resultado de teste e também é exibido um gráfico de Tendência de resultados de teste, conforme a **Figura 14**.

Automatizando a execução de Jobs

Até agora, toda execução de Jobs que fizemos foi através de um processo de acionamento manual, com certeza esse não é o cenário ideal para o dia a dia. Podemos resolver essa questão de duas maneiras (e inclusive combiná-las), a primeira é através de agendamento de execução de Jobs e a segunda é através de scripts de hook do SVN.

Nas configurações de um Job, existe a opção "Trigger the builds>Construir periodicamente", ao marcar essa opção será disponibilizado um campo para digitar as configurações de agenda. Para quem está familiarizado com o Linux, os comandos de agendamento de Jobs do Jenkins seguem o padrão Cron (com pequenas diferenças), por exemplo, para programar um Job para ser executado de hora em hora, basta utilizar o texto: "1 * * * *" que significa que será executado "no minuto 1 de toda hora, todo dia, todo mês e todo dia da semana".

Para não executar um Job de forma desnecessária, é possível condicionar sua execução a uma possível alteração no repositório, para que isso seja possível, basta realizar a mesma configuração na opção "Consultar periodicamente o SCM" e descartar a opção "Construir periodicamente".

A outra opção é utilizar os Hooks do SVN. Os Hooks são rotinas a serem executadas entre eventos de um repositório do SVN. Por exemplo, é possível programar uma rotina para ser executada logo após um Commit em um determinado repositório, em termos técnicos estaríamos configurando um Hook de Post-Commit. No ambiente Windows os Hooks podem ser tanto um executável como um script .bat, enquanto que no ambiente Unix poderia ser um script python, um script shell, um binário C e etc.

O Jenkins possui uma API REST própria, que entre outras coisas, permite que, via HTTP-GET seja possível realizar a construção (Build) de um projeto. A URL desta API segue o seguinte Template: "http://[servidor_jenkins]/job/[nome_do_job]/build". Faremos proveito deste recurso para que, quando um Commit seja disparado em nosso repositório, seja acionado o Build do Job no Jenkins.

Para nos auxiliar neste processo, usaremos o aplicativo WGET (seção **Links**). Este é um aplicativo Free extremamente simples que dentre outras coisas, executa requisições HTTP por linha de comando.

Com o WGET disponível no servidor, vamos finalmente programar o nosso Hook, no VisualSVN Server (**Figura 15**), sendo assim, clique

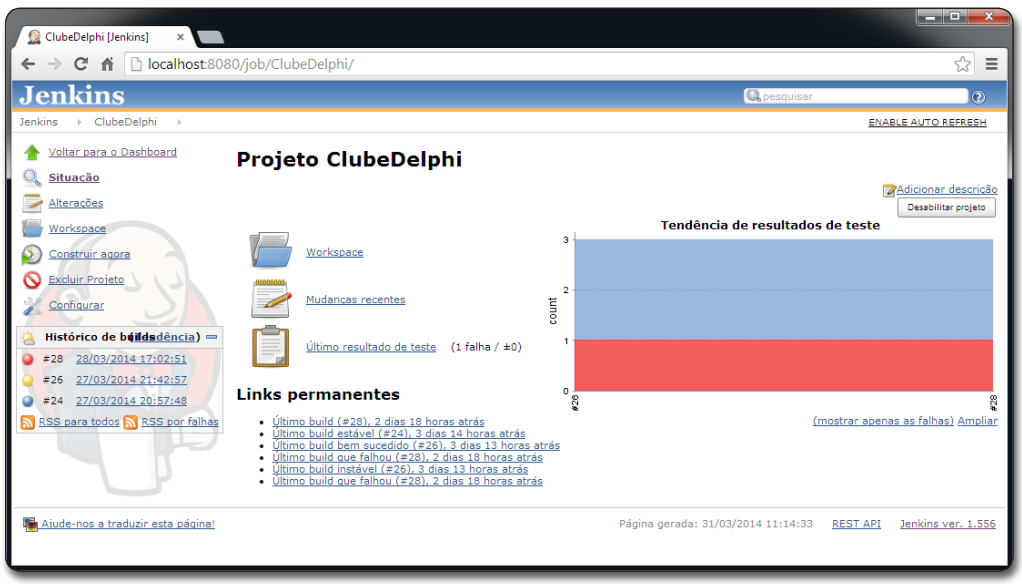


Figura 14. Plugin NUnit em ação

Listagem 6. Projeto de testes atualizado

```
01 program ClubeDelphiTests;

02 {$IFDEF CONSOLE_TESTRUNNER}
03 {$APPTYPE CONSOLE}
04 {$ENDIF}

05 uses
06   TestFrameWork,
07   XMLTestRunner2,
08   GUITestRunner,
09   TestUntDesconto in 'TestUntDesconto.pas',
10   untDesconto in '..\untDesconto.pas';

11 {$R *.RES}

12 var
13   Results: TTestResult;

14 begin
15   if ParamStr(1) = 'jenkins' then
16   begin
17     Results := XMLTestRunner2.RunRegisteredTests('DUnitResult.xml');
18     try
19       if Results.FailureCount + Results.ErrorCount = 0 then
20         ExitCode := 0
21       else
22         ExitCode := 1;
23     finally
24       Results.Free;
25     end;
26   end
27   else
28     GUITestRunner.RunRegisteredTests;
29 end.
```

de direita no repositório e depois selecione a opção "Todas as tarefas -> Manage Hooks".

Será apresentada uma lista com todos os possíveis Hooks. Utilizaremos o Post-Commit Hooks, hook executado logo após o Commit neste repositório. Marque o Hook e depois clique em editar. Respeitando os caminhos configurados em seu ambiente, configure o comando da **Listagem 7** como sendo o comando do Hook.

O parâmetro –b indica que o WGET irá realizar a requisição em Background. Sendo assim o processo foi finalizado, agora ao realizar qualquer Commit em seu repositório, automaticamente será iniciado o processo de Build do seu Job no Jenkins.

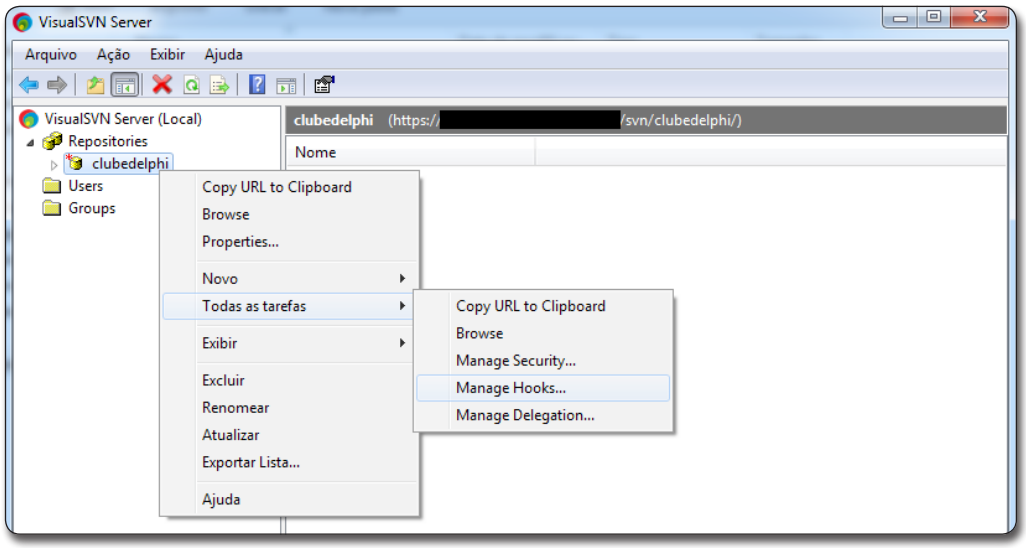


Figura 15. Manage Hooks

Listagem 7. Post-Commit Hook

```
C:/Projetos/wget.exe -b http://localhost:8080/job/ClubeDelphi/build
```

Pensando nos benefícios

As possibilidades dentro da integração contínua são diversas, sendo um dos principais pilares no processo de ALM (Application Life Management). Em integrações manuais, o dia a dia de uma equipe que trabalha em um projeto relativamente mais sofisticado se torna cada vez mais complicada ao entrarem em processos de geração de Build, testes, montagens e etc. Imagine realizar a montagem de um pacote resultante de mais de 20 Commits, se ocorrer um erro durante os testes, seria fácil descobrir o responsável? Seria fácil retomar o raciocínio de mais de cinco Commits atrás?

A adoção da integração contínua colabora para diminuir os riscos de um projeto, a ponto que, ao final de cada dia, é possível se certificar que toda evolução o projeto foi implementada, testada, integrada e possivelmente publicada em um ambiente de testes ou até mesmo de produção, fazendo o que é conhecido como entrega contínua.

Links:

VisualSVN
http://www.visualsvn.com/

Jenkins
http://jenkins-ci.org

Projeto Open Source DUnit
http://dunit.sourceforge.net/.

Download manual do Plugin MSBuild
https://github.com/jenkinsci/msbuild-plugin/releases

Plugin EnvInject
https://github.com/jenkinsci/envinject-plugin/releases

Plugin NUnit
https://github.com/jenkinsci/nunit-plugin/releases

Unit TestRunner2.pas
http://cc.embarcadero.com/Item/28239

WGET
https://www.gnu.org/software/wget/

Integração Contínua
http://martinfowler.com/articles/continuousIntegration.html

Autor

Diego Garcia
drgarcia1986@gmail.com
http://drgarcia1986.wordpress.com
@drgarcia1986
Analista/Desenvolvedor Delphi desde 2008, bacharel em Ciência da Computação e entusiasta de metodologias ágeis e engenharia de software. Atualmente escreve artigos sobre Delphi, Object Pascal e Firebird em seu blog e em publicações especializadas.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

EntityDAC: Ferramenta de Mapeamento Objeto-Relacional – Parte 1

Este artigo faz parte de um curso

O desenvolvimento de aplicações de banco de dados pode ser considerado uma das abordagens mais tradicionais no contexto Delphi. Logo, toda sua prática e conjuntura se apresentam de uma forma bastante natural junto à comunidade, por meio de uma abordagem baseada no desenvolvimento estruturado, adicionado de aspectos orientado a objetos. Em complemento a este cenário, está o desenvolvimento puramente orientado a objetos, em que todo o contexto de negócio e programação está baseado nos conceitos de classes e objetos.

Devido à extensão do assunto, todo o contexto será dividido em duas partes. Nesta primeira será realizada toda a conceituação inicial pertinente, suficiente para a devida iniciação prática neste que pode ser considerado um novo cenário de desenvolvimento de aplicações Delphi.

Fique por Dentro

O Mapeamento Objeto-Relacional é uma técnica tida como fundamental no desenvolvimento Orientado a Objetos de aplicações de banco de dados. Sua finalidade principal é reduzir toda a impedância resultante do uso da POO com bancos de dados relacionais, possibilitando, por exemplo, que uma tabela tenha sua representação na aplicação na forma de um elemento da programação, tal qual uma classe. Em vista disso, diversas ferramentas são disponibilizadas no mercado com o intuito de facilitar o uso desta técnica, automatizando pontos das ações práticas necessárias. Dentre elas está o recém-lançado EntityDAC, da empresa Devart, voltado exclusivamente ao contexto Delphi e cuja intenção principal é tornar este tipo de abordagem facilitada o bastante a ponto de incentivar seu uso por desenvolvedores de todos os níveis.

Conceitos Iniciais

Banco de Dados

Conceitualmente, o termo banco de dados (ou base de dados) é utilizado para se designar uma estrutura especial, refletida numa coleção organizada de informações, as quais são tratadas como dados, em meio ao desenvolvimento de software. Todas as tratativas, incluindo a efetiva operação de toda esta estrutura, ficam a cargo de um software também tido como especial, denominado Sistema de Gerenciamento de Banco de Dados, ou simplesmente SGBD. Seguindo por esta linha a forma com que as informações são tratadas e armazenadas na estrutura acabam por determinar tipos diferentes de bancos de dados. Tomando partido do cenário recente do desenvolvimento de software, dois tipos acabam por se destacar: relacional e orientado a objetos.

De forma sucinta, um banco de dados relacional armazena seus dados na forma de tabelas enquanto que um banco de dados orientado a objetos mantêm suas informações na forma de objetos. Em se tratando do desenvolvimento Delphi, os bancos de dados relacionais são os que se mostram presentes em grande parte das aplicações que envolvem dados. Sem deixar de citar, Oracle, MS SQL Server, InterBase e Firebird são alguns exemplos de bancos relacionais, enquanto que Cache e db4o são exemplos de bancos OO (Orientado a Objetos).

Impedância

Em meio ao cenário recente de software, é correto afirmar que grande parte das informações que transitam por aplicações são mantidas em bancos de dados. Assim como no Delphi, outras linguagens e tecnologias também têm nos bancos de dados relacionais sua principal opção de armazenamento na construção de aplicações que envolvam dados. Logo, numa abordagem primada pelo uso da POO (Programação Orientada a Objetos), o uso de objetos de negócio (ver **BOX 1**), que transitam pela aplicação em substituição aos dados singulares de aplicações construídas a partir de uma abordagem "não-OO", faz com que surjam determinadas exigências naturais. A principal delas se dá pelo uso

de um banco de dados condizente ao modelo de programação, tal como um banco de dados orientado a objetos. Em vista disso, nestes casos, o uso de um banco de dados relacional faz com que uma impedância genuína venha à tona, formalizando o seguinte questionamento: "Como salvar os objetos da aplicação nas tabelas do banco de dados?".

Por exemplo, uma das principais diferenças entre classes e dados relacionais é o conceito de unicidade. Em um banco relacional essa unicidade é garantida através de uma chave primária, mas objetos não possuem e não conhecem a definição de chave primária, eles são únicos por seu endereçamento de memória. Contudo, para minimizar esse disparate algumas técnicas são empregadas, como a técnica dos OIDs.

BOX 1. Objetos de negócio

Objetos de negócio ou entidades de negócio são classes que armazenam valores e os expõe através de propriedades. Eles contêm e gerenciam dados do negócio utilizado por um aplicativo, provendo acesso a esses dados e funcionalidades relacionadas. Entidades de negócio também podem validar os dados contidos nelas e até mesmo encapsular regras para assegurar consistência e integridade quanto às regras e comportamento do negócio envolvido. Além do mais que, modelas e escolher entidades de negócio apropriadas é extremamente importante em um sistema orientado a objetos, compondo sua camada de negócios ou domínio. A **Figura 1** exibe as possíveis camadas que um software orientado a objetos pode possuir.

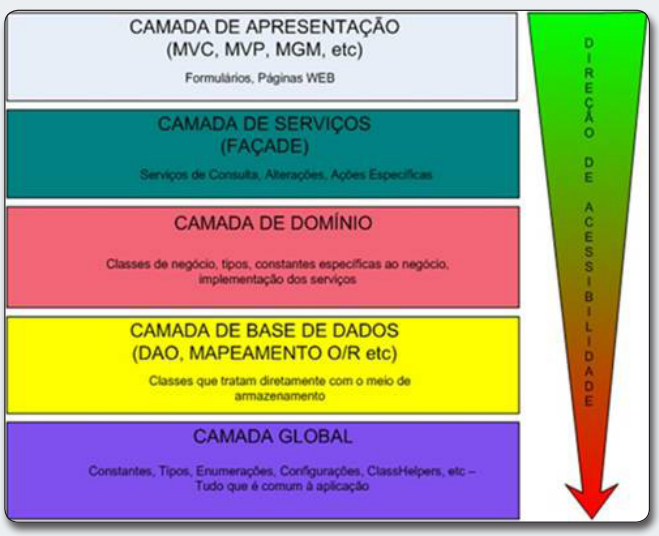


Figura 1. Camadas de um software Orientado a Objetos

Temos que manter um paralelo em alguns aspectos aos bancos de dados relacionais, a exemplo disso temos os OIDs, que são identificadores de objetos.

Como nossos objetos serão salvos em um banco relacional devemos disponibilizar uma forma de encontrá-los. Um OID identifica um único objeto em meio a tantos outros, ele representa a "chave-primária" do nosso banco relacional.

O uso de OIDs facilita a busca de objetos e também o relacionamento entre os objetos. Isso porque quando todas as tabelas utilizam o mesmo tipo de coluna para manter suas chaves, o desenvolvimento de código genérico para lidar com elas é facilitado. Existem algumas considerações sobre as características de um OID.

Um OID não deve ter significado para as regras de negócio: Existe um fato que pode passar despercebido ao modelarmos nossas tabelas. Qualquer coluna de uma tabela que tem significado para as regras do negócio, pode sofrer mudanças, isso porque as regras de negócio podem mudar. Colocar campos que contém significado como chaves é erro que deve ser evitado. Imaginem a situação onde para identificar uma venda é utilizado como chave primária um número que representa a Nota Fiscal. Inicialmente o número da nota fiscal contém apenas números, porém mudam-se as regras de negócio e agora o número de uma NF pode conter letras também. O tipo da coluna que armazena esse valor deverá ser mudado em todas as tabelas, seja onde é chave primária seja onde é chave estrangeira. Uma simples mudança de tipo torna-se um pesadelo. Sendo assim um OID não deve ter significado algum para os negócios, ele deve ser apenas um número sequencial que identifica um único objeto em meio a outros;

Nível de unicidade de um OID: Um outro fator que deve ser considerado ao utilizar OIDs está em sua unicidade, ele pode ser único para o universo todo de objetos, ou pode ser único apenas em meio à classe que pertence. Vamos explicar melhor. Você pode ter, por exemplo, um objeto Aluno que recebe um OID número 5400. Esse número é único em meio a todos os registros do banco de dados. Não existe outro registro com esse OID. Agora, o número 5400 pode também ser único apenas entre os objetos de tipo Aluno, existirão outros registros com esse número, porém pertencerão a outras tabelas/classes. Ao utilizar OIDs deve-se escolher entre essas duas opções, levando em consideração que se escolher a primeira, a coluna que conterá o valor do OID deverá suportar inteiros longos, isso porque você poderá ter bilhões, trilhões de registros em todo o banco de dados.

A técnica de utilização de OIDs é uma de várias outras que são empregadas durante o Mapeamento Objeto-Relacional.

Mapeamento Objeto-Relacional

Em meio ao desenvolvimento, o Mapeamento Objeto-Relacional é comumente tratado como ORM, numa alusão ao termo em inglês Object-Relational Mapping. Conceitualmente pode ser definido como sendo uma técnica de programação utilizada para diminuir pontualmente a impedância existente entre o uso da Programação Orientada a Objetos e bancos de dados relacionais. Para isso, em suma, cada tabela do banco é então espelhada em uma classe na aplicação, tendo suas colunas representadas na forma de propriedades dessa entidade. Seguindo por esta linha, cada registro de uma tabela será representado por uma instância da classe correspondente. Indo além, em cenários como este, todo o trabalho de recuperação e persistência de dados se dá através de uma interface única, substituindo o uso de comandos SQL.

Em razão do oneroso trabalho que é a implementação manual de um mapeamento objeto-relacional, ao longo do tempo, diversos frameworks foram concebidos com o intuito maior de tornar todo

este processo mais facilitado, automatizando diversos pontos do contexto. Apenas para citar, Entity Framework, Hibernate e NHibernate são alguns exemplos populares, relacionados a outras tecnologias, enquanto que DORM, TMS Aurelius e EntityDAC, são exclusivos do Delphi.

EnityDAC

Como definição, o EntityDAC é um produto comercial da empresa Devart e pode ser brevemente definido como sendo uma poderosa ferramenta ORM para Delphi, com suporte a LINQ (ver **BOX 2**).

BOX 2. LINQ

Language Integrated Query (Consulta Integrada à Linguagem), ou simplesmente LINQ, é um componente do Microsoft .NET Framework que provê recursos de consultas a fontes de dados a partir de um modelo unificado, cuja escrita se baseia na própria sintaxe da linguagem de programação (C# e Visual Basic). Logo, esta abordagem substitui o uso de uma linguagem mais especializada, tal como a SQL para banco de dados. Isso provê ao desenvolvedor, entre outras coisas, uma independência com relação às nuances existentes a cada tipo de fonte de dados, prevalecendo o aprendizado a uma única tecnologia, o LINQ.

IDE

O EntityDAC é um produto essencialmente voltado ao RAD Studio da Embarcadero, que sinaliza para as principais ferramentas de desenvolvimento da empresa. Logo, todo seu suporte é previsto para Delphi e C++ Builder, a partir da versão 2007. De forma ilustrativa, este acaba por englobar os IDEs do Delphi XE5 e C++ Builder XE5. Seguindo por esta linha, a seguir é exibida a listagem oficial completa dos IDEs da Embarcadero, compatíveis com o EntityDAC:

- RAD Studio 2007 (Delphi 2007 e C++ Builder 2007);
- RAD Studio 2009 com Update 3 (Delphi 2009 e C++ Builder 2009);
- RAD Studio 2010 (Delphi 2010 e C++ Builder 2010);
- RAD Studio XE (Delphi XE e C++ Builder XE);
- RAD Studio XE2 com Update 4/Hotfix 1 (Delphi XE2 e C++ Builder XE2);
- RAD Studio XE3 com Update 2 (Delphi XE3 e C++ Builder XE3);
- RAD Studio XE4 (Delphi XE4 e C++ Builder XE4);
- RAD Studio XE5 com Update 2 (Delphi XE5 e C++ Builder XE5);

Target Platforms

Tendo em vista o recente acréscimo do suporte a múltiplas plataformas no Delphi, iniciado oficialmente ainda na versão XE2, resultou na adição de um novo recurso em seu IDE, denominado Target Plataforms. De forma breve, tal recurso é refletido em um novo nó presente no gerenciador de projetos (Project Manager) da ferramenta, onde são listadas todas as possíveis plataformas de destino do projeto corrente, com um indicativo da selecionada. Atualmente, na versão XE5, são cinco as opções de direcionamento: Windows-32 bit, Windows 64-bit, Mac OS X, iOS e Android. Dentre estas, neste primeiro momento, o EntityDAC prevê o suporte efetivo às três primeiras, se estabelecendo com uma boa alternativa para uso na construção de aplicações Desktop.

Componentes de acesso a dados

Em sua estrutura de composição o EntityDAC possibilita o trabalho com componentes de acesso a dados para determinados fins. Neste sentido, duas são as opções de integração, que acabam por envolver tanto componentes nativos quanto componentes de sua própria fabricante, a Devart. Olhando pelo lado nativo, ou seja, do conjunto de componentes de acesso a dados que efetivamente acompanham o próprio Delphi, é prevista a integração com todos eles, do BDE ao recente FireDAC, sendo portanto cinco alternativas: ADO, BDE, IBX, dbExpress e FireDAC. Na mesma medida, os principais grupos de componentes de acesso a dados da Devart também são suportados: UniDAC, ODAC, MyDAC, IBDAC, SDAC, PgDAC e LiteDAC.

Edições do EntityDAC

Assim como já é de praxe em produtos comerciais, o EntityDAC é dividido em edições. Mesmo que ainda em caráter Beta (até o momento da escrita deste artigo), é prevista sua segmentação em três edições, especificamente Express Edition, Standard Edition e Professional Edition. Isso faz com que o produto seja disponibilizado ao grande público de diferentes formas, variando em termos de quantidade de recursos suportados, o que acaba por impactar diretamente em seu custo de aquisição. Nominalmente, são vários os recursos comuns a todas as distribuições do EntityDAC. A seguir são listados alguns dos principais, dada sua área de atuação no contexto:

- Mapeamento: Code-mapped Entities, Attribute-mapped Entities, XML-mapped Entities, Attribute-mapped Objects;
- Metadados: Ordinal Meta Types, Generators e One-To-Many;
- Data Context: Get Single Entity, Get Entity List, Save Entity, Direct SQL Execution, Deferred Relation Load, Submit/Reject All Changes;
- LINQ: LINQ queries in runtime;
- Binding: Runtime Live Bindings;
- Devart Data Providers: UniDAC Data Provider, ODAC Data Provider, SDAC Data Provider, MyDAC Data Provider, IBDAC Data Provider, PgDAC Data Provider e LiteDAC Provider;
- Standard Data Providers: (ADO Data Provider, BDE Data Provider, IBX Data Provider, dbExpress Data Provider e FireDAC Data Provider);

Dentre as três opções disponíveis, Express Edition é considerada a edição de entrada do produto, uma vez que contempla uma gama enxuta de features, o que acaba por envolver somente os recursos tidos como básicos. Até mesmo por este fato, sua distribuição se dá de forma gratuita. Visual ORM Model Designer, Complex Meta Types, Create Database, Drop Database, LINQ Code Insight e Design-time Live Bindings são os nomes de alguns dos recursos ausentes nesta edição. Por fim, o EntityDAC Express Edition

também não provê qualquer tipo de componente Data-Aware (ex: EntityDataSet, EntityTable e EntityQuery), pertencentes ao nicho do produto. Em vista de tudo isso, esta edição é direcionada a casos de estudo, experimentação e pequenos trabalhos com o produto.

Na mesma medida, a edição Standard inclui todos os recursos apresentados pela edição Express, com o adicional de funcionalidades pontuais nas áreas de Designer (Visual ORM Model Designer), Mapeamento (Mapping Customization), Metadados (relacionamentos One-To-One e Many-To-Many), Data Context (Create/Drop Database), LINQ (LINQ queries in Design-Time), Binding (Design-Time Live Bindings) e alguns componentes Data-Aware (EntityDataSet e EntityDataSource). Por fim, a edição Professional é tida como a versão completa do produto.

Componentes EntityDAC

Para suas pretensões, o EntityDAC contempla uma série de componentes não-visuais que são então disponibilizados ao desenvolvedor no IDE do Delphi. Sendo assim, imediatamente após sua instalação, dois novos grupos são adicionados à Tool Palette da ferramenta – EntityDAC e EntityDAC Providers – conforme mostrado na **Figura 2**. Em se tratando de um produto novo no mercado, para um ideal aprendizado, faz-se necessária a plena explanação sobre cada um dos elementos presentes nestes agrupamentos.

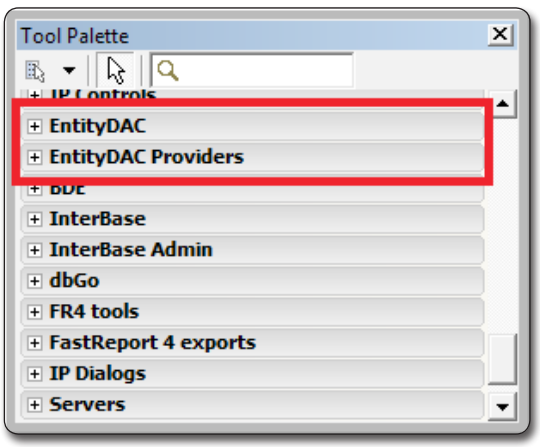


Figura 2. Tool Palette – Grupos de componentes do EntityDAC

TEntityConnection

No contexto histórico do Delphi, os componentes que apresentam o termo "connection" em sua nomenclatura acabam por determinar sua relação direta com a parte de conectividade a uma dada fonte de dados que se relaciona com a aplicação. Logo, com o componente TEntityConnection não é diferente, e sua finalidade principal é prover toda a configuração a diferentes servidores de dados, incluindo um efetivo controle transacional e de sessão.

TEntityModel

O componente TEntityModel provê uma representação visual do modelo de metadados a ser utilizado pelo EntityDAC em tempo de Design. A partir de seu uso é que serão idealmente configurados os componentes relacionadas a dados (DataSet) presentes no projeto de uma aplicação EntityDAC, tais como TEntityTable e TEntityQuery.

TEntityQuery

No Delphi, em aplicações tradicionais de banco de dados, é comum a utilização de um componente Query (ex: TSQLQuery) para a efetiva recuperação dos dados de uma ou mais tabelas do banco de dados relacionado, por meio de uma instrução SQL. Isso faz com que os dados fiquem disponíveis à aplicação, podendo então ser exibidos em controles Data-Aware, através de um objeto de ligação do tipo TDataSource. De forma semelhante, o EntityDAC também contempla um componente Query a ser utilizado na construção de aplicações de banco de dados, que é apresentado como TEntityQuery. Logo, este também se liga a um TDataSource como forma de apresentar suas informações em tela, através de controles Data-Aware. Todavia, num comparativo com a abordagem "clássica", são notadas duas importantes diferenças nesta abordagem com o EntityDAC. A primeira diferença existente fica por conta dos dados, que apesar de se manterem condizentes com as informações das tabelas do banco de dados, são recuperados na forma de instâncias de entidades de negócio, seguindo o paradigma da Orientação a Objetos. Em complemento, o outro contraste notório diz respeito ao comando utilizado na recuperação das informações, que aqui se dá pelo uso de instruções LINQ.

TEntityTable

O dbExpress, um dos frameworks nativos até então mais utilizados na construção de aplicações Delphi de banco de dados, apresenta em sua biblioteca de componentes os elementos TSQLQuery e TSQLTable que, em suma, se assemelham pelo fato de estarem diretamente relacionados à manipulação dos dados provenientes das tabelas do banco de dados. A diferença fundamental existente entre eles é que o primeiro, conforme já mencionado, pode estar relacionado aos dados de uma ou N tabelas, por meio do uso de instruções SQL. Já o segundo, TSQLTable, se liga essencialmente a apenas uma única tabela, não prevendo o uso de qualquer tipo de instrução de comando. No EntityDAC, este cenário então se repete, uma vez que sua biblioteca também apresenta um componente Query (TEntityQuery), já apresentado anteriormente, bem como um componente Table (TEntityTable), que recupera e atualiza as entidades de um único tipo (meta-type), não provendo o uso de instruções LINQ.

TEntityDataSet e TEntityDataSource

TEntityDataSet é um componente previsto em todas as edições do produto, cuja funcionalidade acaba por resumir alguns aspectos das tratativas relacionadas a dados. Na prática, ele é tido como um elemento RunTime only, ou seja, que só pode ser usado em tempo de execução, sendo capaz de manter uma lista de entidades provenientes dos registros de uma fonte arbitrária. Já o TEntityDataSource refere-se a um componente que, conforme seu próprio nome sugere,

se assemelha ao tradicional TDataSource. Sendo assim, sua função básica é prover uma interface comum para a devida ligação entre controles visuais e componentes de dados, que se refletem em controles Data-Aware e componentes DataSet, respectivamente.

TEntityContext

A ideologia imposta pelo EntityDAC acaba por determinar o uso de elementos compostos de dados, tratadas como entidades, em substituição ao uso de dados singulares simples, provenientes das aplicações de banco de dados tradicionais. Em vista disso, TEntityContext é o nome da classe de um componente especial presente na biblioteca do EntityDAC, cuja responsabilidade é então gerenciar as entidades presentes no contexto da aplicação. Na prática, isso acaba por envolver os processos relacionados à manipulação dos dados relacionados ao banco, mais precisamente sobre as entidades, que resultam nas tradicionais funções CRUD (Create-Retrieve-Update-Delete): criação (Create), recuperação (Retrieve), atualização (Update) e exclusão (Delete). Adicionalmente, o componente TEntityContext mantém as entidades utilizadas em cache, otimizando uma futura reutilização. Da mesma forma, o componente é hábil o suficiente para destruir automaticamente toda e qualquer entidade armazenada, que não faz mais parte do contexto do software, ou seja, que não é mais utilizada pela aplicação.

TDBXDataProvider, TIBXDataProvider, TBDEDataProvider, TADODataProvider e TFireDACDataProvider

O grupo EntityDAC Providers apresenta os elementos responsáveis pela ligação de outros grupos de componentes específicos de dados ao provedor de dados (Data Provider) natural de uma aplicação EntityDAC. Especificamente, cada elemento está relacionado a uma biblioteca nativa de componentes de dados, conforme a seguir:

- TIBXDataProvider: IBX (InterBase eXpress);
- TDBXDataProvider: DBX (dbExpress);
- TBDEDataProvider: BDE;
- TADODataProvider: ADO;
- TFireDACDataProvider: FireDAC;

Instalação

A instalação do EntityDAC segue o formato de Wizard, tal como mostra a **Figura 3**, promovido pelo padrão Windows NNF (Next-Next-Finish). Dessa forma, todo o processo ocorre de forma bastante natural e intuitiva ao usuário, sem nenhuma observação especial. Apenas como informativo, dois pontos merecem certo destaque. O primeiro é uma particularidade apresentada pelo próprio instalador do EntityDAC que exige que nenhuma instância do Delphi esteja aberta (dentre aquelas suportadas) no momento da instalação, precavendo-se de qualquer eventual falha de integração com o ambiente. Caso esta condição seja violada, o próprio instalador se encarrega de mostrar uma mensagem de alerta (**Figura 4**), que impede a sequência do processo. O segundo ponto diz respeito ao suporte previsto pelo produto às diferentes versões do IDE do Delphi, que aqui se dá a partir de um executável-instalador único. Assim, a seleção pelo IDE condizente acontece já no processo de instalação, conforme mostra a **Figura 5**. Obviamente, dentre as alternativas, somente são habilitadas aquelas detectadas no ambiente.

Figura 3. EntityDAC – Wizard de instalação

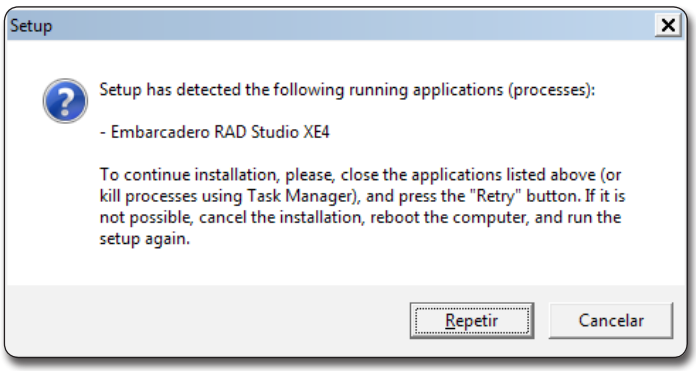


Figura 4. EntityDAC – Wizard de Instalação – Mensagem de aviso

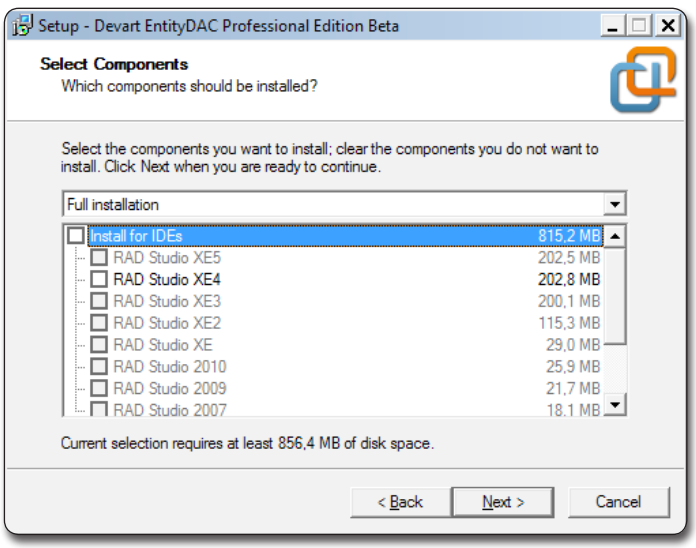


Figura 5. EntityDAC – Instalação - Opções de IDE

Em termos de versão e edição do EntityDAC, o presente artigo toma como base o EntityDAC Professional Edition 1.0.3 Beta, que é a opção mais recente disponível no momento da escrita deste trabalho.

Primeiro passos com o EntityDAC - Conceitos

Uma vez instalado, o uso inicial do EntityDAC se dá pelo menu homônimo e exclusivo, incorporado ao IDE do Delphi, conforme mostrado na **Figura 6**. Dentre as opções disponibilizadas, apenas para mencionar, EntityDAC Help e EntityDAC FAQ servem como guia de instruções, tanto para o novo utilizador quanto ao já habitual, sanando dúvidas pontuais. Em complemento, a opção EntityDeveloper é quem efetivamente dá início às atividades com o framework, provendo acesso direto à estrutura de trabalho do EntityDAC. Logo, assim que é acionada, esta opção traz à tona um novo ambiente de desenvolvimento, denominado Entity Developer (**Figura 7**).

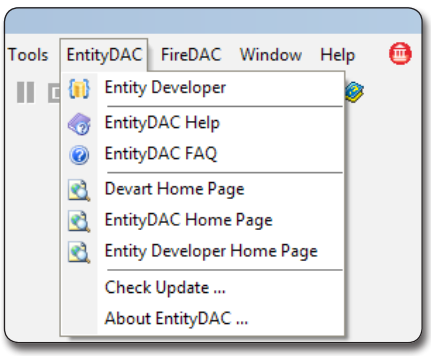


Figura 6. IDE do Delphi – menu EntityDAC

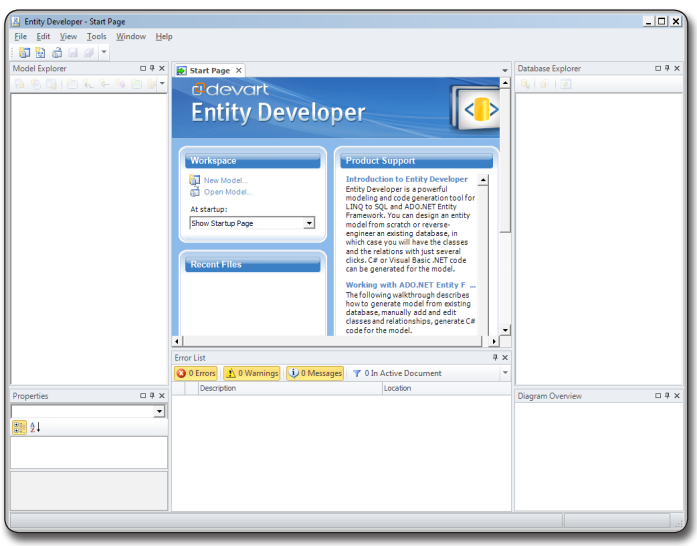


Figura 7. Entity Developer

De uma forma geral, a construção de uma aplicação baseada no uso de um framework para mapeamento objeto-relacional acaba por estabelecer a criação de um Modelo (Model), como ponto de partida de desenvolvimento. Em suma, é nesta estrutura onde estarão definidas as classes de negócios envolvidas no projeto, dando origem às futuras instâncias de entidades (objetos). Neste ponto, duas abordagens se mostram bastante comuns: a criação de um modelo inicial "do zero" ou o provimento de um modelo a partir de um banco de dados existente. No caso do EntityDAC, estas opções são apresentadas nominalmente como Model First e Database First:

- Model First: determina a criação de um modelo integral, sem se basear em qualquer estrutura (ex: banco de dados) pré-existente. Uma vez criado, torna possível a geração do banco de dados a partir do Modelo;
- Database First: determina a criação de um modelo a partir da estrutura de um banco de dados pré-existente. Logo, para a efetivação do processo, apenas uma conexão ativa com a base é necessária. Pelo fato de se basear em algo pronto (neste caso o banco de dados propriamente dito), esta abordagem tende a ser interpretada como mais simples, se comparada a Model First;

Criando o Modelo

Diante do que foi mencionado, a fim de se ilustrar uma exemplificação prática, será tomada como base a abordagem Database First. Logo, para a criação do Modelo, o próprio ambiente do Entity Developer dispõe de uma opção direcionada, acessível pelo menu File>New Model, ou pelo atalho Ctrl+N. Quando acionada, esta faz a chamada a uma nova janela que, conforme mostrado na **Figura 8**, dispõe de uma série de opções, mais especificamente quatro – Code-mapped entites, Attribute-mapped entities, XML-mapped entities e Attribute-mapped objects – que determinam tipos de modelos pré-definidos. Em suma, todos eles acabam por envolver dois aspectos básicos de mapeamento, relacionados às classes (entidades) e aos seus respectivos metadados. Por conseguinte, a variação sobre as tratativas com relação a estes aspectos é que determina o diferencial substancial existente entre cada tipo de modelo.

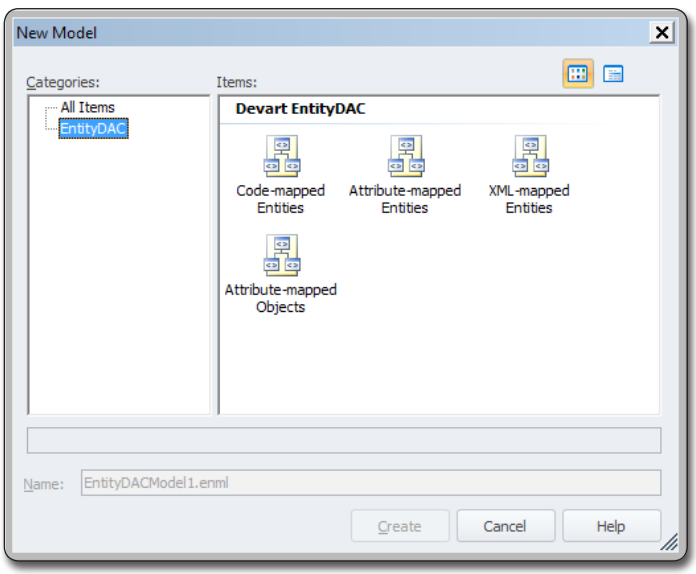


Figura 8. Entity Developer – New Model – Opções de modelo

1. Code-mapped entities (entidades mapeadas por código)
Esta opção determina que para cada entidade detectada no banco de dados será gerada uma classe relacionada no modelo. No EntityDAC, classes como estas são criadas como descendente do tipo TEntity. Além disso, elementos descendentes de TEntity são automaticamente gerenciados pelo contexto de dados do framework. No aspecto prático, os metadados do modelo são gerados em uma unit separada, na forma de um conjunto de classes "especiais", que estarão vinculadas às classes de entidade.

2. Attribute-mapped entities (entidades mapeadas por atributo)
Opção de comportamento similar a anterior, em razão de que as classes que irão compor o modelo serão geradas com base na leitura das entidades detectadas no banco de dados apontado. Todavia, como diferença fundamental, aqui não há a geração de classes complementares de metadados, ao invés disso, as próprias classes de entidade são marcadas com atributos especiais, que funcionam como anotações. A partir deles, o próprio framework se encarrega de gerar os metadados condizentes em runtime. Por este fato, a opção "Attribute-mapped entities" se mostra como uma alternativa elegante, no provimento do modelo a ser utilizado pelo projeto da aplicação.

3. XML-mapped entities (entidades mapeadas por XML)
Neste tipo de modelo, novamente somente as classes de entidade são geradas a partir do banco de dados. Logo, o mapeamento do modelo, ou seja, sua parte de metadados é direcionada a um arquivo externo, do tipo XML. Por este fato, esta opção tende a deixar todo o código envolvido mais "limpo", livre de qualquer tipo de anotação, bem como mais enxuto, uma vez que nenhuma classe adicional de metadados é mantida. Por outro lado, a opção XML-mapped entities mostra-se inicialmente um tanto quanto mais trabalhosa que as demais abordagens, pelo fato da necessidade da criação e manutenção de um arquivo externo relacionado ao modelo.

4. Attribute-mapped objects (objetos mapeados por atributo)
Até aqui, todas as opções apresentadas, relacionadas aos tipos pré-definidos de modelo disponíveis, acabam por contextualizar classes descendentes do tipo TEntity, próprio do framework EntityDAC. Em complemento a este cenário, "Attribute-mapped objects" substitui tal ancestral por um tipo nativo, mais especificamente TObject, que é classe ancestral raiz de todas as classes do Delphi. A opção pelo uso deste tipo de modelo acaba por determinar um código quase que independente de elementos externos (fora do contexto nativo). Por fim, aqui o mapeamento do modelo se dá pelo uso de atributos, tal como no tipo "Attribute-mapped entities".

Diante dos tipos de modelos disponíveis, visando dar uma maior praticidade a este desenvolvimento introdutório, será tomado como base o tipo "Code-mapped entities". Conforme visto, esta opção exime o desenvolvedor de qualquer intervenção manual para consolidar o devido mapeamento objeto-relacional e formação do Model (aqui tratado como ClubeDelphiModel.enml). Por conseguinte, uma vez definido e confirmado o tipo do modelo, o Entity Developer novamente provê uma facilidade, mais precisamente um assistente inteligente (Wizard), denominado Create Model Wizard.

Create Model Wizard

Diante do que foi mencionado, o Wizard ficará então responsável por prover todas as instruções necessárias ao utilizador, a fim de se obter como resultado final o modelo propriamente dito. Já de antemão, vale frisar que Create Model Wizard é sensível ao contexto, o que significa que seus passos variam de acordo com as opções escolhidas pelo próprio usuário. Por esta razão e com o intuito maior de contribuir na fixação de conceitos, a seguir serão explicitadas todas as etapas pertinentes a uma segmentação completa do assistente.

Selecionando a forma com que o modelo será criado

A **Figura 9** ilustra a primeira etapa do assistente, onde deverá ser selecionada a forma com que o modelo será criado, se Database First ou Model First. Essas opções acabam por definir as duas segmentações propostas pelo assistente, logo, a escolha entre uma ou outra envolverá passos distintos. Seguindo a proposta do exemplo, aqui será tomada como base a primeira.

Definindo as propriedades da conexão com o banco de dados

O passo seguinte envolve a definição das propriedades da conexão com o banco de dados, cuja informação principal diz respeito ao Provider a ser utilizado no contexto. Por padrão, são disponibilizados três agrupamentos distintos de provedores: Devart dotConnect (for DB2, MySQL, Oracle, PostgreSQL e SQLite), .NET Framework Data Provider (for Microsoft SQL Server Compact 3.5/4.0, MySQL, Oracle e SQLServer) e Oracle Data Provider. Caso opte pela utilização de um banco de dados diferente dos SGBDs previstos, faz-se necessária a instalação de um Provider adequado. Sendo assim, tomando como base um cenário bastante comum em meio ao desenvolvimento de aplicações Delphi de banco de dados, o uso do SGBD Firebird se mostra bastante tradicional. Logo, sua utilização neste contexto envolve a instalação de um Provider específico, tal como o ".NET Provider for Firebird", disponível de forma oficial e gratuita no próprio site do produto. Uma vez instalado, este fica então devidamente disponível na lista apresentada nesta etapa do assistente (**Figura 10**).

Especificando o conteúdo do modelo

Com a conexão com o banco de dados explicitamente definida a próxima etapa do assistente envolve a especificação do conteúdo do modelo. Para isso, o Create Model Wizard trabalha com duas opções, disponibilizadas como "Generate From Database" e "Empty Model".

A primeira, que pode ser considerada a mais intuitiva, possibilitará que o desenvolvedor escolha, dentre os objetos do banco de dados, aqueles que farão parte do modelo. Já a segunda, conforme seu próprio nome prevê, dará origem a um modelo "em branco", mas ainda plenamente vinculado ao banco de dados, o que torna possível uma eventual e posterior adição de qualquer objeto do banco ao modelo corrente.

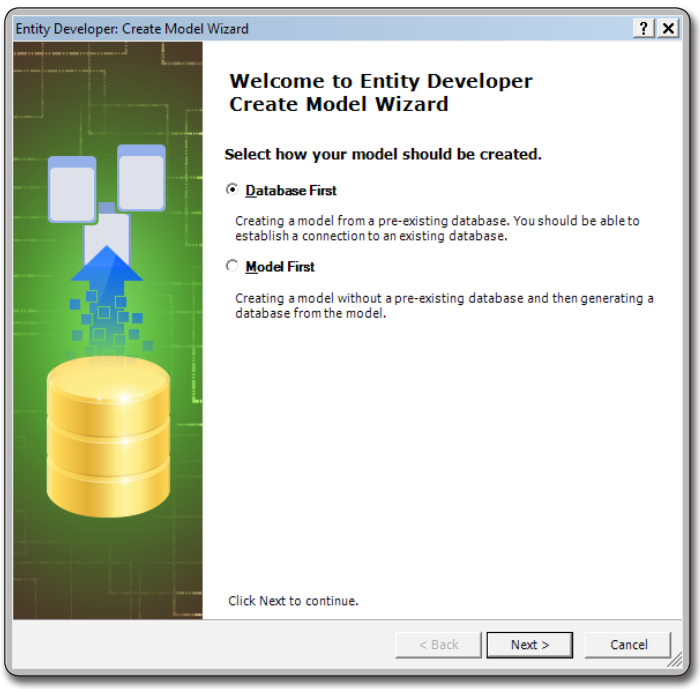


Figura 9. Entity Developer – Create Model Wizard – Forma de criação do modelo

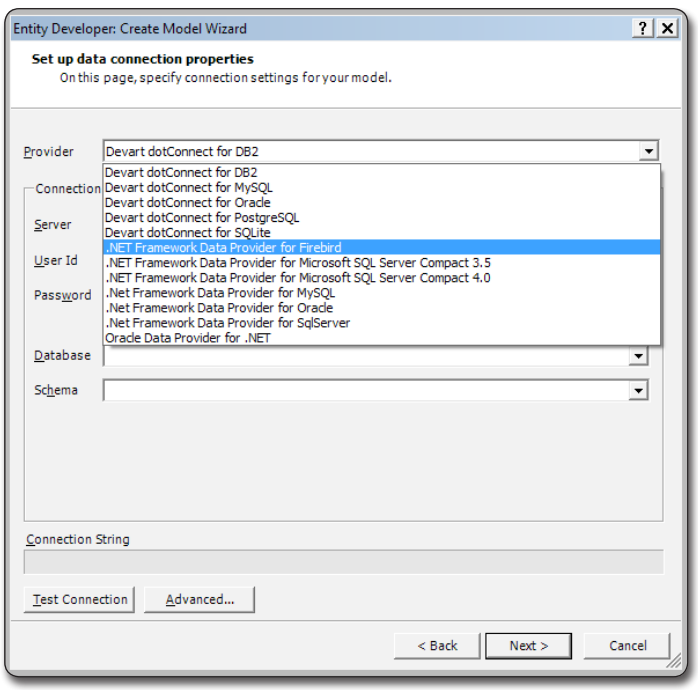


Figura 10. Entity Developer – Create Model Wizard – Propriedades da conexão

Partindo-se do pressuposto da escolha da opção "Generate From Database", a **Figura 11** mostra a tela de seleção dos objetos do banco de dados que darão origem às entidades do modelo, formalizando assim mais uma etapa do assistente. Sem deixar de mencionar, a imagem em questão ilustra as opções de objetos do banco Employee, que acompanha a instalação do Firebird.

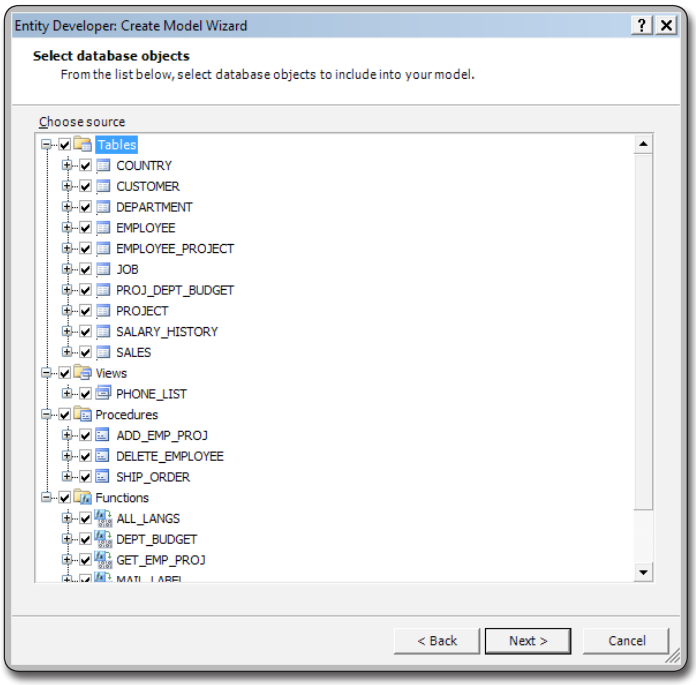


Figura 11. Create Model Wizard – Selecionando os objetos do banco de dados

Regras de nomeação para os elementos do modelo

Na prática, a formação do modelo com base no banco de dados estabelece, basicamente, a transformação das tabelas em entidades (classes), bem como suas colunas em propriedades dessas. Em vista disso, ao gerar tais classes, por padrão, o Entity Developer automaticamente atribui os nomes para os elementos produzidos (classes e propriedades) com base na própria nomeação de origem. Visando dar uma maior abertura ao desenvolvedor, uma das etapas do assistente envolve a definição de uma regra particular de nomeação para as classes e propriedades geradas. Neste momento, é importante ressaltar que este tipo de manipulação não é vital para o processo, logo, numa abordagem inicial, a própria regra de nomeação padrão tende a ser a utilizada. Apenas para ilustrar o que foi mencionado, a **Figura 12** exibe este passo do Wizard.

Definindo as propriedades do modelo

A próxima etapa provida pelo Create Model Wizard, intitulada Model properties (**Figura 13**), possibilita a definição de algumas das propriedades relacionadas ao modelo que está sendo criado. Como exemplo, dentre estas, "Use database comments" define o aproveitamento dos comentários contidos no banco de dados, "Allow nullable String types" permite a criação de propriedades

do tipo String que aceitem valor Null, assim como "Detect Many-to-Many associations" detecta a existência de associações N:N (muitos-para-muitos) baseada nas associações existentes entre as tabelas monitoradas.

Escolhendo o conteúdo do diagrama do modelo

O modelo provido pelo EntityDAC está associado a um diagrama que, em poucas palavras, pode ser definido como sendo sua representação visual. Numa exemplificação mais tradicional, traçando um paralelo próximo, um banco de dados pode estar associado a um diagrama de banco de dados, que nada mais é do que uma representação gráfica da estrutura de seu conteúdo. Logo, neste passo do assistente (**Figura 14**), são providas ao desenvolvedor algumas opções, mais precisamente três, sobre como se dará a formação efetiva do conteúdo deste diagrama no contexto do framework:

- All Entities: opção padrão que estabelece a presença de todas as entidades do modelo no diagrama;
- Split By Databases: opção voltada ao trabalho com múltiplos bancos de dados. Logo, as entidades dos modelos são agrupadas por banco, sendo que para cada grupo deste, um novo diagrama é criado;
- Custom: opção que provê uma maior liberdade ao desenvolvedor, uma vez que possibilita a seleção das entidades que farão parte do diagrama.

Mediante as opções apresentadas, esta exemplificação inicial mantém a linhagem padrão provida pelo Wizard através da seleção de "All Entities".

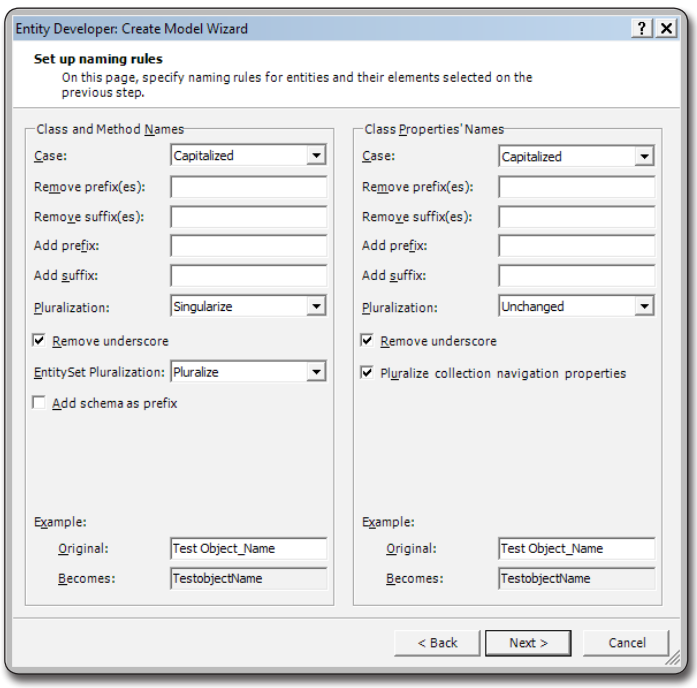


Figura 12. Create Model Wizard – Definindo regras de nomeação

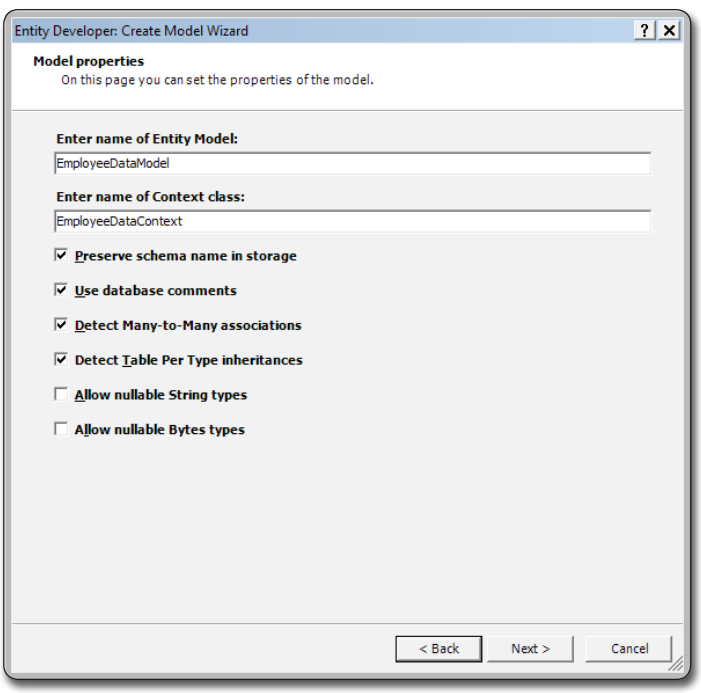


Figura 13. Create Model Wizard – Definindo as propriedades do modelo

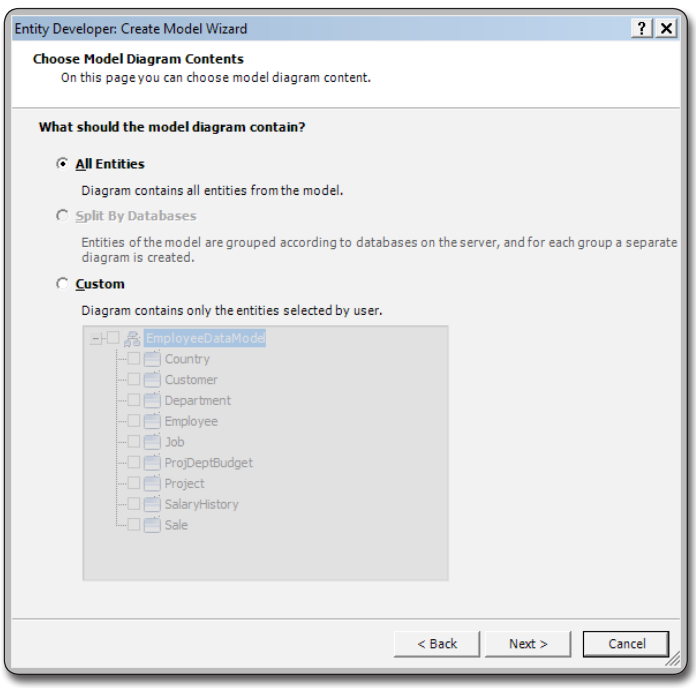


Figura 14. Create Model Wizard – Escolhendo o conteúdo do diagrama do modelo

Escolhendo o Code Generation Template

"Choose Code Generation Templates" é o título da última etapa do assistente que estabelece a seleção e configuração do Template de geração de código do modelo, aqui tratado como Code Generation Template. Por padrão, o Template relacionado ao tipo de modelo escolhido já vem selecionado, tal como mostra a **Figura 15**. Neste caso, a adoção do tipo de modelo Code-mapped Entities fez com que seja automaticamente habilitado o Template

Code-mapped Entity Classes. Além deste, Attribute-mapped Entity Classes, XML-mapped Entity Classes e Attribute-mapped Objects são as outras opções disponíveis.

Por conseguinte, dentre as configurações possíveis e relevantes neste cenário, está a definição dos nomes dos arquivos que serão gerados pelo framework. De acordo com os conceitos apresentados, a quantidade de arquivos envolvidos está diretamente relacionada ao tipo de modelo que se está trabalhando. No caso do Code-mapped Entities, são três units (.pas):

- Unit "entities": arquivo de código Delphi que contém as declarações das classes de entidade do modelo;
- Unit "metadata": arquivo de código Delphi que contém as declarações das classes de metadados relacionadas ao modelo;
- Unit "LINQ": arquivo de código Delphi que contém as declarações de classes utilitárias tidas como especiais, que possibilitarão a escrita de consultas no estilo LINQ.

Para a formação dos nomes de cada arquivo destes o Wizard traz como padrão os indicadores "Entities", "MetaData" e "Linq", respectivamente, que então receberão como prefixo o próprio nome do modelo. Tomando como base o modelo atual, denominado ClubeDelphiModel, o resultado disso seria a produção dos seguintes arquivos: ClubeDelphiModelEntities.pas,

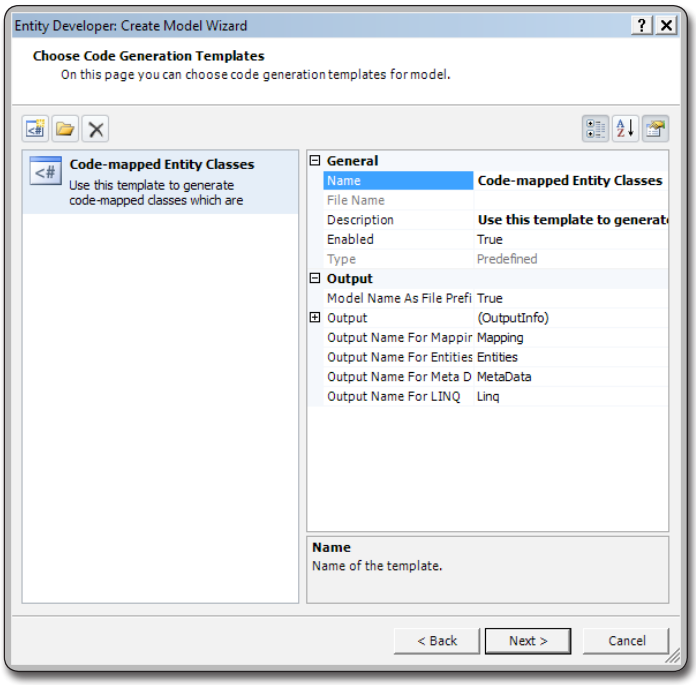


Figura 15. Create Model Wizard – Escolhendo o Code Generation Template

DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior portal para desenvolvedores da América Latina!

20 mil posts

430 mil cadastrados

10 milhões de page-views por mês

http://www.devmedia.com.br/

DEVMEDIA

ClubeDelphiModelMetaData.pas e ClubeDelphiModelLinq.pas. Apenas para mencionar, toda esta prefixação é estabelecida pela propriedade "Model Name As File Prefix".

Ao confirmar este passo no botão Next, uma nova tela é trazida à tona (**Figura 16**), apresentando uma mensagem final, cujos dizeres informam a criação com sucesso do modelo. Todavia, a plena efetivação da criação física do modelo e de seu diagrama associado ocorre através do botão Finish, presente nesta última tela. Feito isso, enfim o Entity Developer expõe de forma visual o diagrama do modelo criado. De forma ilustrativa, a **Figura 17** expõe uma amostragem a partir do modelo provindo do banco de dados Employee do Firebird, abordado até aqui.

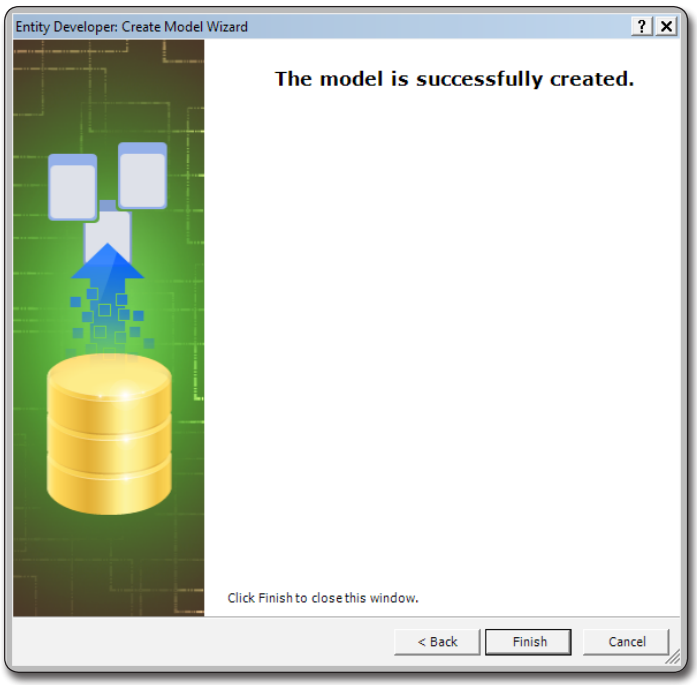


Figura 16. Create Model Wizard – Mensagem final de sucesso

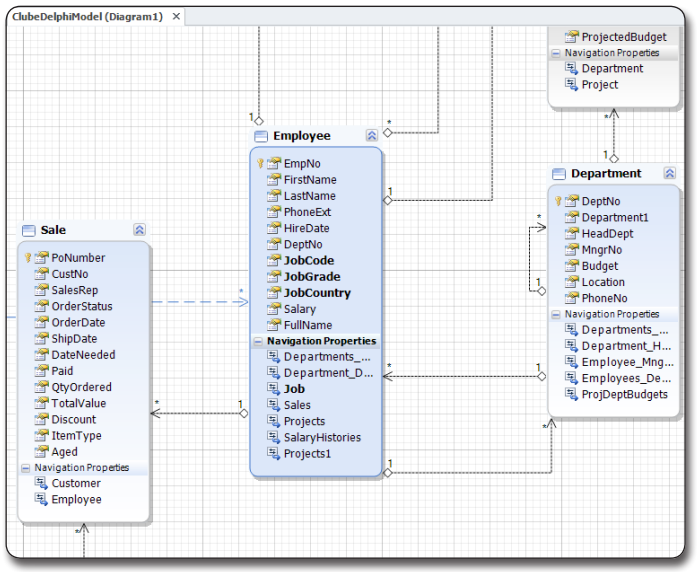


Figura 17. Entity Developer – Diagram

O presente artigo trouxe como foco um novo produto inserido recentemente no cenário Delphi, mas que promete e tem a intenção de se fixar como uma poderosa ferramenta auxiliar no cotidiano dos desenvolvedores da comunidade. Isto porque sua função principal tende a explorar um aspecto que foge ao tradicional do desenvolvimento de aplicações Delphi de banco de dados, que se refere a uma abordagem pautada essencialmente nos aspectos da POO (Programação Orientada a Objetos). Assim, numa citação simples, ao invés dos simples dados presentes numa abordagem tradicional, aqui se trabalha com entidades de negócio. Logo, somente esta mudança acaba por estabelecer outro paradigma na construção do software. Dentre estes aspectos, o EntityDAC vem para sanar a questão da impedância existente entre a Orientação a Objeto e o uso de bancos de dados relacionais, caracterizando-se como uma poderosa ferramenta de mapeamento objeto-relacional.

Autor

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ e pós-graduado em Engenharia de Componentes. Atua como Analista Programador Delphi há 9 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor. Site pessoal: www.fabriciokawata.com.br

Links:

RAD Studio XE5 – Página oficial do produto
http://www.embarcadero.com/products/rad-studio

RAD Studio XE5 – Download Trial
https://downloads.embarcadero.com/free/rad_studio

EntityDAC – Página oficial do produto
http://www.devart.com/entitydac

Firebird – Página oficial do projeto
http://www.firebirdsql.org

.NET Provider for Firebird
http://www.firebirdsql.org/en/net-provider

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Metodologia Ágil no Delphi XE6

Mantendo suas aplicações de forma ágil

Este artigo é do tipo mentoring

saiba mais: www.devmedia.com.br/mentoring-saibamais

Cenário

Neste artigo veremos alguns recursos de programação orientada a objetos com Delphi, como herança, abstração e polimorfismo, aplicados em um exemplo prático. Através do uso de boas práticas, veremos como estes fundamentos podem tornar um software mais fácil de ser mantido e evoluído, usando ainda algumas técnicas de desenvolvimento ágil associada a design patterns, como a refatoração rumo a padrões de projeto. Veremos como resolver problemas comuns encontrados em código fonte (bad smells, ou "maus cheiros"), usando uma abordagem de desenvolvimento corretiva e evolutiva. Uma introdução ao desenvolvimento ágil evolutivo é apresentada, como forma de unir estas técnicas de orientação a objeto com as técnicas atuais de engenharia de software, demonstradas em um exemplo prático com Delphi XE6.

Metodologias clássicas de engenharia de software são muito burocráticas, focadas em planejamento, dão ênfase em contratos fixos, documentação excessiva, vasto planejamento antecipado (prematuro). Essa abordagem não condiz mais com a realidade dos projetos na área de tecnologia, onde novos requisitos são criados, modificados ou mesmo removidos após um planejamento inicial. Já as metodologias ágeis de desenvolvimento são evolutivas por natureza, organizam o trabalho de forma interativa, realizado em pequenas partes, com pequenas entregas, ao invés de um único release enorme. Dão maior ênfase na colaboração e comunicação aberta e fluente entre stakeholders, clientes e membros do time, minimizando as chances de insucesso em projetos. Os princípios básicos da abordagem ágil sugerem que indivíduos e interações são mais importantes que processos e ferramentas, trabalho deve ser feito mais focado no software em si que em uma documentação abrangente, colaboração do cliente mais que negociação de contrato e, principalmente, responder a mudanças mais que seguir um plano. Em software, estima-se que até 50% de custos são gastos com design e planejamento. Uma metodologia ágil como a Extreme Programming prega cinco valores principais: comunicação, feedback, simplicidade, coragem e respeito, com foco maior em disciplina do que processos. Já Scrum prega um desenvolvimento ágil baseado em iterações chamadas sprints e reuniões de acompanhamento frequentes (meetings). Aplicações reais precisam continuamente ser evoluídas, novos requisitos implementados, melhorias devem ser feitas no projeto existente (design), algumas vezes é necessário mudar de tecnologia ou plataforma, corrigir bugs, otimizar e melhorar a performance. Para muitas empresas, o foco na construção de um software é torná-lo fácil de ser mantido ao longo do tempo. Nesse sentido, é necessário tomar cuidado com bons princípios de design, como criar componentes com baixo acoplamento, reutilizáveis, que garantam que alterações em componentes não afetam outros, que objetos tenham responsabilidades únicas, separadas, partes facilmente substituíveis, complexidade encapsulada. Esses são princípios básicos da programação orientada a objetos que, vistos dentro de um contexto de uma abordagem ágil evolutiva, vão tornar o sistema de software mais fácil de testar, evoluir, manter e entender.

Padrões de Projeto (Design Patterns) ajudam a criar bons designs de software, já o uso de Refatoração contínua garante código claro, limpo e funcional em equipes que programam de forma mais ágil, sem se preocupar em um primeiro momento com muitos aspectos de design. Padrões de Projeto são soluções reutilizáveis para problemas recorrentes no desenvolvimento orientado a objetos. Aplicar padrões antecipadamente em um projeto requer mais tempo e aumenta custos, ao mesmo tempo em que aumenta sua complexidade. Requisitos são modificados, criados ou mesmo eliminados conforme projetos evoluem. Tentar aplicar padrões antecipadamente para prever essas mudanças é um problema conhecido como excesso de engenharia. Já projeto em escassez reduz custos, pois o sistema de software é criado rapidamente devido à falta de tempo, conhecimento ou necessidade de se adicionar funcionalidades em curto prazo. Ao longo do tempo, armazena-se um débito de projeto, que torna o sistema de software muito difícil de

ser mantido e evoluído. É nesse contexto que entram as refatorações, como uma forma de melhorar o projeto existente sem alterar seu comportamento externo observável.

Refatoração é o aspecto-chave para um projeto evolutivo. Nesse caso, desenvolvimento ágil, evolutivo, incremental, apoiado em refatorações e testes, é uma excelente receita para garantir o sucesso do projeto, seja ela para um startup ou uma grande corporação.

Este artigo vai mostrar como aplicar estes conceitos em um exemplo prático com Delphi XE6. Serão demonstrados os principais fundamentos da orientação a objetos (ver **BOX 1**) em uma aplicação que simula notificação de clientes de uma determinada companhia, por exemplo, uma operadora de telefonia, que precisa avisar seus clientes sobre cobranças, promoções e novidades. Além dos fundamentos da orientação a objetos, são demonstradas importantes técnicas como programação para interfaces, padrões, anti-patterns, bad smells, uso contínuo de refatoração para um desenvolvimento ágil com código sempre limpo e funcional, além do uso de refatoração rumo a padrões de projeto.

BOX 1. Os quatro pilares da Programação Orientada a Objetos

i) Abstração: capacidade de representar conceitos do domínio do problema, ressaltando apenas o que for relevante para a aplicação em questão. A definição da classe em Delphi atende a esse requisito, pois ela é um modelo limitado de alguma entidade real, que implementa apenas as características que são necessárias para a aplicação sendo desenvolvida em um dado contexto.

ii) Encapsulamento: a linguagem deve permitir a definição de módulos autossuficientes, possibilitando a implementação do conceito de ocultação de informação (information hiding), ou seja, esconder os detalhes internos. O uso dos especificadores de visibilidade (private e protected, por exemplo) e o comportamento dos objetos em Delphi reforçam esse princípio, principalmente se o desenvolvedor usar propriedades nas classes, como forma de acesso aos campos internos.

iii) Herança: mecanismo que permite a definição de uma hierarquia de classes, com o objetivo de reutilizar características já definidas e estendê-las em níveis maiores de detalhes ou especialização. O Delphi implementa um mecanismo eficiente de herança simples (apenas uma classe ancestral – não há suporte para herança múltipla de classes).

iv) Polimorfismo: classes descendentes podem implementar de forma diferente um conjunto de operações comuns.

Criando a aplicação

No Delphi XE6, vamos iniciar uma nova aplicação Delphi do tipo Console (**Figura 1**), não vamos trabalhar com uma interface de usuário em específico (Web, Windows, Mobile), pois as técnicas aqui apresentadas valem para qualquer tecnologia. Desta forma, utilizar uma aplicação de Console permitirá focarmos mais na aplicação dos fundamentos e padrões da orientação a objetos do que nos elementos de interface de usuário. Este exemplo também pode ser feito em versões anteriores do Delphi, como XE2, XE3, XE4 ou XE5. O nome da aplicação será "SistemaNotificacao".

A **Listagem 1** mostra a versão 1.0 do sistema. Aqui lemos algumas informações da tela, para envio da mensagem por e-mail, como o código do cliente (destinatário), e a mensagem a ser enviada. Para fins didáticos, não vamos incluir aqui código para uso de classes de envio de e-mails (usando Indy por exemplo), o que foge o escopo do tema principal do artigo, então estamos simplesmente simulando o envio mostrando uma mensagem no console com o código do cliente a mensagem.

O primeiro bad smell que pode ser visto no código é que estamos misturando lógica de negócio (o envio do e-mail) com tratamento de interface de usuário (Console). Se fosse necessário usar o código de envio da mensagem por e-mail em outra parte do sistema, este teria que ser duplicado, o que é um dos piores maus cheiros em código. Em outras palavras, estamos comprometendo a reutilização de código de negócio, visto que está codificado diretamente na camada de apresentação, que poderia ser uma aplicação VCL Forms, FireMonkey ou Web.

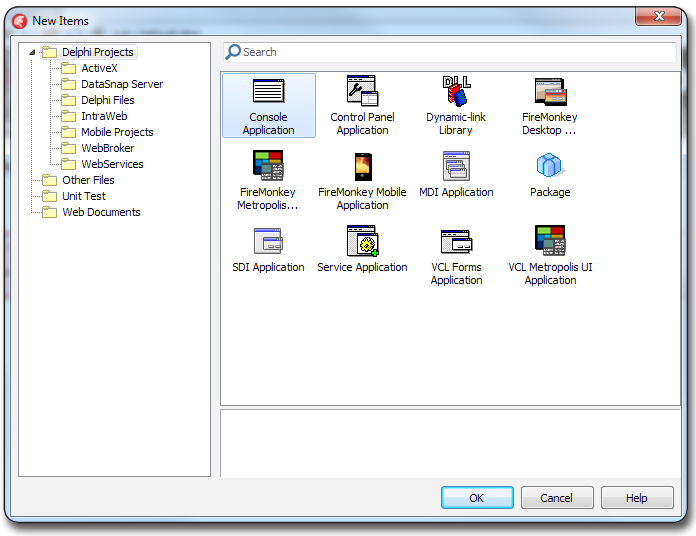


Figura 1. Criando uma aplicação Console no Delphi XE6

A primeira versão da aplicação está pronta e pode ser entregue no prazo estipulado, mas não está com a melhor estrutura adequada para comportar possíveis mudanças. Não aplicamos boas práticas de programação orientada a objetos e engenharia de software, como separação de responsabilidades, nem padrões de projeto. Ela não está pronta para mudanças. Como programadores não são videntes, não se pode sair aplicando um padrão de projeto de forma antecipada, causando complexidade desnecessária. O requisito único e inicial do sistema era enviar notificações por e-mails para clientes sobre faturas e promoções, e isso foi feito dentro do escopo, no prazo estipulado, com os recursos orçados. A **Figura 2** mostra a aplicação em funcionamento.

Uma premissa básica de qualquer sistema de software é a mudança. Seja para incorporar uma nova funcionalidade, ou corrigir bugs existentes, evoluir para uma nova plataforma. Nosso cliente, depois de receber a versão 1.0 do sistema, decide que agora alguns dos seus clientes precisam ser notificados através de mensagens SMS, e não apenas por e-mail, pois alguns deles não estavam recebendo as mensagens corretamente por estarem caindo na caixa de spam.

A **Listagem 2** mostra a versão 2.0 do sistema. Aqui estamos lendo as informações da tela, para envio da mensagem que agora pode ser enviada por e-mail ou por SMS. Note que, devido ao curto prazo que a equipe teve para implementar a nova funcionalidade, novamente

não houve uma preocupação em acomodar essa nova funcionalidade de uma forma mais adequada, usando uma separação de responsabilidade, ou usando um padrão de projeto. Simplesmente incluímos um teste condicional para verificar se o operador do sistema escolheu enviar a mensagem por e-mail ou por SMS, e de acordo com essa escolha, a operação de envio é realizada.

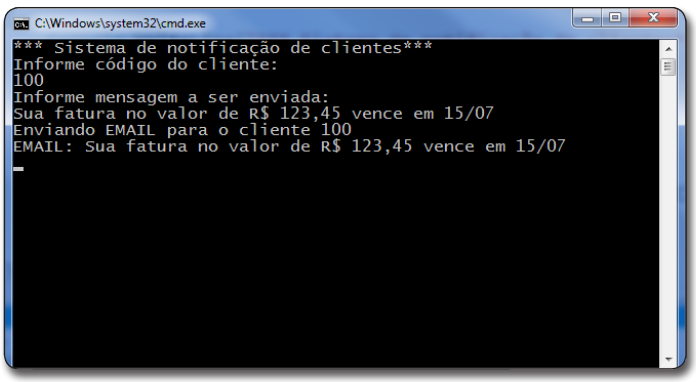


Figura 2. Aplicação notificando cliente por e-mail sobre vencimento de fatura

Listagem 1. Sistema de Notificação versão 1.0 segundo requisitos iniciais levantados

```
program SistemaNotificacao;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils;

var
  CodCliente: string;
  Mensagem: string;
begin
  WriteLn('*** Sistema de notificação de clientes***');
  WriteLn('Informe código do cliente:');
  ReadLn(CodCliente);
  WriteLn('Informe mensagem a ser enviada:');
  Readln(Mensagem);
  // Simula envio de um email
  WriteLn(format('Enviando EMAIL para o cliente %s',[CodCliente]));
  WriteLn(format('EMAIL: %s',[Mensagem]));
  Readln;
end.
```

A versão 2.0 do sistema foi entregue dentro do prazo, porém, está com uma série de problemas, ou bad smells. Primeiro, como já comentamos, não estamos separando responsabilidades: o que é interface de usuário, como é feito o envio de e-mails e mensagens SMS. O código está impossível de ser reutilizado, por exemplo, em uma aplicação Web ou Mobile. Além disso, estamos usando um "tipo código", também chamado de Magic Number ("Número mágico"). Esse é um bad smell muito clássico, onde temos no código um número que fica difícil de saber seu significado, ou pelo menos, exige maior cuidado do desenvolvedor que precisar dar manutenção no código. Aqui, o número "1" está sendo usado para representar o que deveria ser um tipo chamado EMail, e "2" para representar um SMS. Outro bad smell que podemos notar aqui é a "Obsessão Primitiva", muito comum em programadores iniciantes em orientação a objetos. Normalmente, eles tendem a trabalhar com tipos primitivos da linguagem, como na passagem de parâmetros e valor de retorno de funções. São tipos como bool, integer e string. Além disso, usar um número (aqui em formato string) para identificar um tipo poderia extrapolar o domínio da variável, por exemplo, é perfeitamente possível colocar um "3", que não representa nenhum dos tipos de envio que estamos trabalhando (SMS e Email). Além disso, o compilador não consegue verificar esse tipo antes da execução.

Outro bad smell claro que podemos ver no código é o "Método Longo", ou seja, ele está muito grande em termos de linhas de código. Alguns autores sugerem que um método deve ter, no máximo, 10 linhas. Alguns sugerem até cinco. Outro bad smell presente no exemplo é o chamado "Sentenças Condicionais" (IFs, Elses, Cases). Este tipo de construção deixa o código complexo de entender, pois o programador que precisar modificá-lo deve ser capaz de interpretar cada uma das possibilidades a fim de entender o que o algoritmo está fazendo. Além disso, este tipo de teste tende a duplicar código, caso esse teste de envio precisasse ser feito em mais de um local. E finalmente, usar expressões condicionais em excesso dificulta testes unitários e a cobertura de testes, já que todas as possibilidades devem ser avaliadas, gerando vários possíveis caminhos. Em se tratando de manutenção de software, existe uma métrica chamada "Complexidade Ciclomática", que visa justamente medir o quanto uma classe está difícil de entender devido a sentenças condicionais. Tal métrica pode ser verificada no menu QA Metrics ativando-se o suporte a modelagem UML no IDE do Delphi XE6.

O que temos na versão 2.0 é o chamado "Débito de Projeto". Ela foi entregue rapidamente para cumprir o prazo, ninguém na equipe se preocupou se aquela seria a melhor forma de comportar o novo requisito. Não houve uma preocupação por aplicar um padrão de projeto, ou mesmo, princípios básicos da orientação a objetos. Chamamos de débito porque, assim como um cartão de crédito que não tem sua fatura paga, vai aumentar os juros ao longo do tempo, e esses juros vão se acumulando, até que o débito está tão grande que é impossível pagá-lo. A equipe não realizou uma tarefa vital entre o release 1 e 2, o processo de refactoring (refatoração).

Refatoração

Refatoração é uma importante técnica de desenvolvimento ágil. Refatorar significa mudar o comportamento interno do sistema de software sem mudar seu comportamento externo observável. Refatoração pode ajudar a tornar o código mais legível para ser entendido por outros programadores, facilitando a manutenção corretiva futura. Pode ajudar a melhorar sua estrutura interna a fim de que novos requisitos possam ser acomodados de forma mais adequada, sem introduzir bugs. Refatoração pode ajudar a tornar o código mais reutilizável. Permite que bugs sejam encontrados de forma mais rápida, visto que o código está sempre limpo e organizado. O problema da refatoração é que é muito

difícil explicar sua vantagem para gerentes. O que dizer sobre uma técnica que não agregará valor nenhum ao cliente final, em termos de funcionalidades, consumindo tempo da equipe de desenvolvimento? Lembre-se, a refatoração não muda o comportamento externo observável. O que a maioria dos gerentes não enxerga é que a refatoração deve ser um hábito contínuo durante o desenvolvimento do software, para que não ocorra o débito de projeto. A equipe gasta mais tempo aplicando padrões e boas práticas entre os releases, porém o software estará sempre bem estruturado, fácil de entender, manter e evoluir.

Listagem 2. Sistema de Notificação versão 2.0, com débito de projeto

```
program SistemaNotificacao;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils;

var
  CodCliente: string;
  Mensagem: string;
  ModoEnvio: string;
begin
  WriteLn('*** Sistema de notificação de clientes***');
  WriteLn('Informe código do cliente:');
  ReadLn(CodCliente);
  WriteLn('Informe mensagem a ser enviada:');
  Readln(Mensagem);
  WriteLn('Digite opção de envio, 1-EMAIL ou 2-SMS:');
  Readln(ModoEnvio);
  if (ModoEnvio = '1') then
  begin
    // Simula envio de um email
    WriteLn(format('Enviando EMAIL para o cliente %s',[CodCliente]));
    WriteLn(format('EMAIL: %s',[Mensagem]));
  end
  else
    if (ModoEnvio = '2') then
    begin
      // Simula envio de um SMS
      WriteLn(format('Enviando SMS para o cliente %s',[CodCliente]));
      WriteLn(format('SMS: %s',[Mensagem]));
    end;
  Readln;
end.
```

Em nosso exemplo, imagine agora que na versão 3.0 do sistema nosso cliente solicite a inclusão de uma nova funcionalidade, o envio de mensagens por VoiceMail (gravação de voz), visto que alguns usuários da operadora não estão conseguindo receber as devidas notificações nem por e-mail nem por SMS. A equipe decide, novamente, por não aplicar boas práticas ou padrões, ou mesmo refatorar o código existente, entregando a versão 3.0 como mostra a **Listagem 3**. Foi incluído mais um teste condicional, para verificar a nova opção de envio. Aqui já podemos citar mais um clássico bad smell, uma "Classe Deus" (God Class). Uma classe "Deus" é aquela que faz tudo, está muito sobrecarregada de responsabilidades, faz mais daquilo que inicialmente foi projetada para fazer. É impossível reutilizar seu código, pois tudo nela depende dela mesmo. O seu índice de manutenção (outra métrica que pode ser mensurada pelo Delphi XE6) está comprometido. Ou seja, temos um débito de projeto enorme a ser pago.

Vamos voltar no tempo. Quando entregamos a versão 1.0, não foi especificado que o sistema um dia teria que comportar várias formas de envio. Uma premissa básica de todo sistema de software é a mudança, tudo bem. Mas que tipo de mudança? Programadores devem codificar para hoje, não são videntes. Quando a versão 2.0 foi entregue, houve o que chamamos de "Escassez de Engenharia". Sem padrões, sem classe, não foi feita sequer uma refatoração. Em contrapartida, se a equipe tivesse preparado a versão 1.0 com um padrão de projeto que permitisse tornar o código mais fácil de ser mantido e evoluído, teria causado o que chamamos de "Excesso de Engenharia". Ou seja, aplicar um padrão de projeto de forma antecipada, sem saber se ele realmente vai ser necessário. Afinal, quem disse, na versão 1.0, que o cliente poderia pedir um novo tipo de envio na versão 2.0? Se o sistema funcionasse para o resto de sua vida funcionando simplesmente enviando notificações por e-mail, talvez tivéssemos criado uma arquitetura complexa totalmente desnecessária.

Então, qual é a solução se não podemos programar nem com escassez nem com excesso? Simples, aplicando-se constantemente refatorações entre os releases, de modo a não deixar se acumular um grande débito de projeto, impossível de ser pago. Imagine se o cliente solicitasse, na versão 4.0, uma nova funcionalidade, para incluir um novo tipo de envio (iMessage, WhatsApp?). Nesse ponto do sistema, a maioria das empresas já está com um código repleto de bad smells, difícil de evoluir, de entender e de manter. O **caos** já está generalizado, e em algum momento a equipe já pensa em escrever tudo do **zero**. Nesse ponto é comum ocorrer uma **rotatividade** de profissionais na equipe, o que agrava ainda mais a situação. Um novo requisito que era para ser implementado e entregue em poucos dias, leva semanas. Bugs começam a aparecer sem explicação alguma. Um bug corrigido causa 10 novos bugs em outros lugares.

Melhorando o Design do Projeto

Agora vamos aplicar a boa prática de refatoração. Ela poderia ser aplicada antes da entrega da versão 2.0 do sistema, mas como já estamos na versão 3.0, decidimos melhorar a estrutura do nosso código antes que o caos se instaure na versão 4.0 e a equipe inteira deixa a empresa. O próprio IDE do Delphi XE6 oferece suporte automático para a atividade de refatoração. Você pode selecionar cada um dos blocos de código que faz operações diferentes de envio e escolher a opção Extract Method (**Figura 3**). Aqui vamos criar um método bem nomeado para cada uma das operações (envio de e-mail, SMS e VoiceMail). A **Listagem 4** mostra como ficou a aplicação após estas refatorações. Foram criados os métodos EnviarEmail, EnviarSMS e EnviarVoiceMail. Cada um deles possui sua própria implementação. Eles então são executados de acordo com a escolha do usuário.

Se executarmos a aplicação, veremos que ela continua funcionando exatamente da mesma forma, porém nosso código agora está mais bem estruturado.

Listagem 3. Sistema de Notificação versão 3.0, com enorme débito de projeto

```
program SistemaNotificacao;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils;

var
  CodCliente: string;
  Mensagem: string;
  ModoEnvio: string;
begin
  WriteLn('*** Sistema de notificação de clientes***');
  WriteLn('Informe código do cliente:');
  ReadLn(CodCliente);
  WriteLn('Informe mensagem a ser enviada:');
  Readln(Mensagem);
  WriteLn('Digite opção de envio, 1-EMAIL, 2-SMS, 3-Voice:');
  Readln(ModoEnvio);
  if (ModoEnvio = '1') then
  begin
    // Simula envio de um email
    WriteLn(format('Enviando EMAIL para o cliente %s',[CodCliente]));
    WriteLn(format('EMAIL: %s',[Mensagem]));
  end
  else
    if (ModoEnvio = '2') then
    begin
      // Simula envio de um SMS
      WriteLn(format('Enviando SMS para o cliente %s',[CodCliente]));
      WriteLn(format('SMS: %s',[Mensagem]));
    end
    else
      if (ModoEnvio = '3') then
      begin
        // Simula envio de um VoiceMail
        WriteLn(format('Enviando VoiceMail para o cliente %s',[CodCliente]));
        WriteLn(format('VoiceMail: %s',[Mensagem]));
      end;
  ReadLn;
end.
```

Nota

Uma boa prática associada à refatoração é a de aplicar testes unitários para garantir que o comportamento externo observável não foi modificado. Esse é um princípio do TDD – Test Driven Development. Juntos, refatoração e testes podem garantir código sempre limpo e funcional. O IDE do Delphi XE6 também inclui o suporte integrado para criação de testes unitários através do DUnit, incluído no IDE.

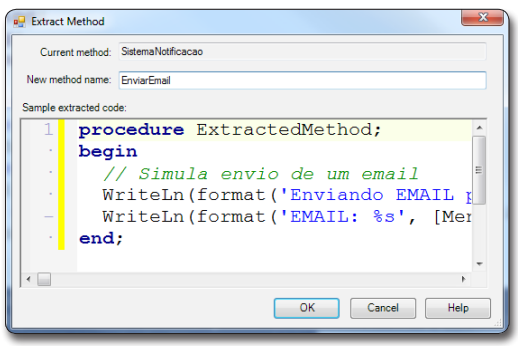


Figura 3. Refatorando com Extract Method

Listagem 4. Sistema de Notificação versão 3.0, refatorado com Extract Method

```
program SistemaNotificacao;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils;

var
  CodCliente: string;
  Mensagem: string;
  ModoEnvio: string;

procedure EnviarEmail(CodCliente: string; Mensagem: string);
begin
  // Simula envio de um email
  WriteLn(format('Enviando EMAIL para o cliente %s',[CodCliente]));
  WriteLn(format('EMAIL: %s',[Mensagem]));
end;

procedure EnviarSMS(CodCliente: string; Mensagem: string);
begin
  // Simula envio de um SMS
  WriteLn(format('Enviando SMS para o cliente %s',[CodCliente]));
  WriteLn(format('SMS: %s',[Mensagem]));
end;

procedure EnviarVoiceMail(CodCliente: string; Mensagem: string);
begin
  // Simula envio de um VoiceMail
  WriteLn(format('Enviando VoiceMail para o cliente %s',[CodCliente]));
  WriteLn(format('VoiceMail: %s',[Mensagem]));
end;

begin
  WriteLn('*** Sistema de notificação de clientes***');
  WriteLn('Informe código do cliente:');
  ReadLn(CodCliente);
  WriteLn('Informe mensagem a ser enviada:');
  Readln(Mensagem);
  WriteLn('Digite opção de envio, 1-EMAIL, 2-SMS, 3-Voice:');
  Readln(ModoEnvio);
  if (ModoEnvio = '1') then
    EnviarEmail(CodCliente,Mensagem)
  else
    if (ModoEnvio = '2') then
      EnviarSMS(CodCliente,Mensagem)
    else
      if (ModoEnvio = '3') then
        EnviarSMS(CodCliente,Mensagem);
  ReadLn;
end.
```

Separando Responsabilidades

A refatoração deve ser feita em pequenos passos, ou o que chamamos de "baby steps" (passo de bebê). Você não efetua um "big refactoring", pois várias mudanças precisariam ser testadas de uma única vez. A regra é, faça uma mudança de cada vez, compile e teste. Nossa próxima refatoração vai ser a Move Method, ou seja, vamos mover os métodos de envio de e-mail, SMS

e VoiceMail para classes distintas, separando responsabilidades. Isso é feito através outra refatoração chamada Extract Class. Dessa forma, criamos três novas classes no sistema, e movemos os códigos relativos a cada uma para sua respectiva classe. Uma classe Email deveria fazer o que? Enviar e-mails, obviamente, e somente isso. Essa não é mais responsabilidade da interface de usuário. Se outra aplicação precisar enviar e-mails, fica muito simples reaproveitar essa lógica.

Nesse ponto, aplicamos a refatoração Rename no nome dos métodos para simplesmente Enviar. Isso porque um método chamado Enviar dentro de uma classe Email é suficiente para identificar sua intenção. Aqui fica outra dica de código limpo, nomeie seus métodos como verbos, deixando clara sua intenção. Métodos, variáveis e expressões bem nomeadas dispensam comentários, pois o código se torna simples de ler por qualquer outro profissional da equipe. Não é necessário incluir comentários em um código que está fácil de ler e entender. Se você encontrar um código muito comentado, significa que ele está difícil de entender, e não deveria ser comentado, deveria ser refatorado. O próprio uso excessivo de comentários é considerado um bad smell.

A **Listagem 5** mostra o código refatorado. O código que cria as classes e chama os métodos Enviar, usando sentenças condicionais, está um pouco complexo de ser entendido, mas não se preocupe com isso agora, será resolvido em uma refatoração adiante. Lembre-se, pequenas refatorações, uma de cada vez.

Nota

"Qualquer tolo pode escrever código que um computador possa entender. Bons programadores escrevem código que humanos podem entender." Martin Fowler.

Abstraindo os Detalhes

Um dos principais fundamentos da orientação a objetos é a abstração, como comentado no início do artigo. Uma das melhores

Listagem 5. Sistema de Notificação versão 3.0, refatorado com Move Method e Extract Class

```
// Programa Principal
program SistemaNotificacao;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils,
  GPauli.Notificacao in 'GPauli.Notificacao.pas';

var
  CodCliente: string;
  Mensagem: string;
  ModoEnvio: string;
begin
  WriteLn('*** Sistema de notificação de clientes***');
  WriteLn('Informe código do cliente:');
  ReadLn(CodCliente);
  WriteLn('Informe mensagem a ser enviada:');
  ReadIn(Mensagem);
  WriteLn('Digite opção de envio, 1-EMAIL, 2-SMS, 3-Voice:');
  ReadIn(ModoEnvio);
  if (ModoEnvio = '1') then
    Email.Create().Enviar(CodCliente,Mensagem)
  else
    if (ModoEnvio = '2') then
      SMS.Create().Enviar(CodCliente,Mensagem)
    else
      if (ModoEnvio = '3') then
        VoiceMail.Create().Enviar(CodCliente,Mensagem);
  ReadIn;
end.

// Unit do Framework de Notificação
unit GPauli.Notificacao;

interface

uses
  System.SysUtils;

type
  Email = class
    public procedure Enviar(CodCliente: string; Mensagem: string);
  end;

  SMS = class
    public procedure Enviar(CodCliente: string; Mensagem: string);
  end;

  VoiceMail = class
    public procedure Enviar(CodCliente: string; Mensagem: string);
  end;

implementation

{ Email }

procedure Email.Enviar(CodCliente: string; Mensagem: string);
begin
  // Simula envio de um email
  WriteLn(format('Enviando EMAIL para o cliente %s',[CodCliente]));
  WriteLn(format('EMAIL: %s',[Mensagem]));
end;

{ SMS }

procedure SMS.Enviar(CodCliente: string; Mensagem: string);
begin
  // Simula envio de um SMS
  WriteLn(format('Enviando SMS para o cliente %s',[CodCliente]));
  WriteLn(format('SMS: %s',[Mensagem]));
end;

{ VoiceMail }

procedure VoiceMail.Enviar(CodCliente: string; Mensagem: string);
begin
  // Simula envio de um VoiceMail
  WriteLn(format('Enviando VoiceMail para o cliente %s',[CodCliente]));
  WriteLn(format('VoiceMail: %s',[Mensagem]));
end;

end.
```

práticas da programação orientada a objetos diz: "Programe para Abstrações / Interfaces". Nesse ponto, nosso sistema está com uma dependência muito grande entre o programa principal, que representa a interface de usuário, das classes que realizam as operações de envio, o que chamamos de classes concretas.

Depender de implementações concretas é um problema, porque tudo muda, essa é uma premissa básica quando se desenvolve software. Certamente hoje não se compra uma TV que não tenha uma entrada HDMI, que permita a mudança de um reprodutor de Blu-ray (por um PS4, por exemplo) sem alterar a interface, o que seria uma tragédia. Imagine ter que enviar um televisor de 52 polegadas de volta para fábrica solicitando que a mesma crie uma nova porta no aparelho para que se possa ligar um novo PS4. Certamente não é dessa forma que funciona na vida real. Na vida real, existem padrões, abstrações e interfaces, que clientes e fornecedores declaram e implementam para que mudanças não tenham impacto na forma como os objetos são relacionados. Imagine se a especificação da interface USB fosse modificada? Bilhões de dispositivos ao redor do mundo precisariam ser modificados. Na vida real, nós sempre pensamos para abstrações, em qualquer situação. Pare, pense e reflita. Se eu pretendo trocar de carro em 2016, eu não digo "vou comprar o modelo XYZ Sport 7.2" e ponto final. Pode ser que o modelo saia de linha, fique muito caro, ou um novo modelo mais moderno o substitua. Eu simplesmente falo que vou trocar de carro e pronto, pois "carro" é uma abstração do modelo, e com isso eu não quebro meu contrato. Fico preparado para mudanças. Outro exemplo: suponha que sua esposa tenha pedido para você comprar um gato de estimação na veterinária até o final do dia, e você está saindo de casa logo pela manhã, com pressa. Você poderia dizer: "Querida, vou encontrar um gato siamês com um olho verde e outro azul e um detalhe no pelo na pata dianteira esquerda, com certeza um dos mais bonitos que se possa encontrar". Chegando na veterinária, você só encontra hamsters, aranhas e cobras. Ou seja, você especificou demais o que iria fazer, e dessa forma, fica difícil seguir o contrato e ficar preparado para mudanças, em termos da orientação a objetos, seguir uma especificação / interface. Ao invés disso, você poderia ter dito: "Ok, trarei um animal!" (pensando na abstração). E dessa forma, esse é seu contrato, você abstraiu mais os detalhes, ficou melhor preparado para mudanças, e se ao final das contas você levar um jacaré para casa, ainda estará seguindo a especificação. Tudo bem que na vida real isso traria alguns problemas, mas na orientação a objetos isso é perfeitamente aceitável e uma das melhores práticas que se pode utilizar.

Assim deveria ser nosso sistema, porém, ele está agora como mostra a **Figura 4**. Supondo que nosso programa possuísse uma referência (associação) para cada um dos tipos concretos na forma de propriedade (no momento estamos instanciando diretamente no bloco de código principal), o diagrama de classes da aplicação seria representado dessa forma. Veja que há uma ligação direta com cada um dos tipos, não com uma abstração, que permita variar o comportamento, aplicando polimorfismo.

Então vamos aplicar uma refatoração chamada Extract Super Class, ou ainda, Extract Interface, no caso da abstração ser "pura"

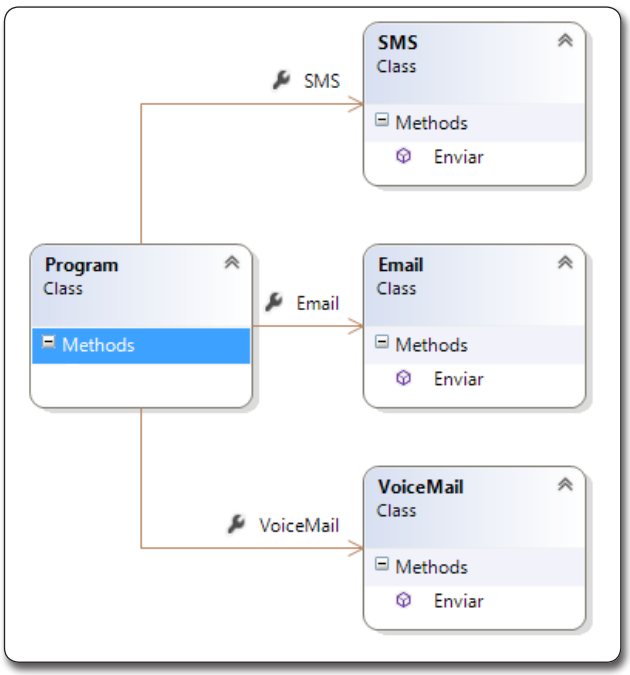


Figura 4. Dependência de classes concretas é um problema que causa forte acoplamento

(totalmente abstrata). Isso porque classes abstratas permitem a inclusão de código comum a classes descendentes, de forma a compartilhar código e reduzir duplicação. Já interfaces são totalmente abstratas, não incluem qualquer tipo de implementação, somente especificação. É uma espécie de contrato entre cliente e classes que proveem determinado serviço ou funcionalidade (as classes concretas, ou produtos). O cliente e produto se comprometem a seguir a especificação, de forma a evitar problemas quando algo mudar, ou evoluir. Ambos seguem um contrato, imutável, de baixo acoplamento, já que interfaces não possuem implementação. Assim, aplicando outro conceito fundamental da orientação a objetos, o polimorfismo, podemos fazer com que nossa classe do programa principal se comunique com a abstração, e em tempo de execução, a correta classe de envio concreta assumirá o lugar da abstração, fornecendo uma forma diferente de envio. Em Delphi, isso pode ser feito através do uso das palavras reservadas virtual + abstract na classe base (ou somente virtual no caso de abstração parcial) e override nos métodos das classes descendentes.

Nota

Para fins didáticos, estamos considerando aqui que o arquivo Program do Delphi é uma "classe", como se representasse, por exemplo, um formulário VCL ou FireMonkey consumindo as classes do framework de notificação.

A **Listagem 6** e **Figura 5** mostram as alterações feitas. Declaramos uma classe abstrata chamada Notificacao, com um método virtual abstrato polimórfico que serve de base para diferentes comportamentos que devem ser sobescritos em classes descendentes (Email, SMS e VoiceMail). O programa principal agora se comunica com uma abstração, declarada através de uma propriedade (associa-

ção) referenciando Notificacao, que em tempo de execução pode receber qualquer classe concreta que implementa esta abstração, por herança. Veja que nesse caso há uma herança de abstração / interface, o que difere de herança de implementações.

Esse tipo de herança, de apenas dois níveis, é ótimo para reduzir acoplamento entre classes, ligando-as através de um ponto leve (interface).

Nota

Uma boa prática é usar herança de "caixa-preta", muito comum em padrões de projeto como Decorator, onde esconde-se uma classe "herdada" como um atributo privado interno, e repassar chamadas a métodos desse objeto interno, por delegação, o que inclusive serve para simular herança múltipla em Delphi, já que se pode compor um objeto a partir de vários outros, e implementar várias interfaces ao invés de herdar de uma única classe base.

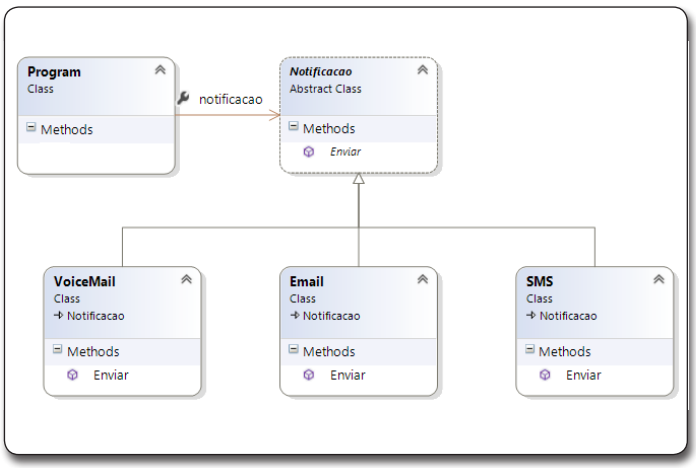


Figura 5. Programando para uma interface / abstração

Listagem 6. Sistema de Notificação versão 3.0, usando abstração e polimofismo

```
// Programa Principal
program SistemaNotificacao;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils,
  GPauli.Notificacao in 'GPauli.Notificacao.pas';

var
  CodCliente: string;
  Mensagem: string;
  ModoEnvio: integer;
  Opcao: TipoNotifacao;
begin
  WriteLn('*** Sistema de notificação de clientes***');
  WriteLn('Informe código do cliente:');
  ReadLn(CodCliente);
  WriteLn('Informe mensagem a ser enviada:');
  Readln(Mensagem);
  WriteLn('Digite opção de envio, 1-EMAIL, 2-SMS, 3-Voice:');
  ReadIn(ModoEnvio);
  Opcao := TipoNotifacao(ModoEnvio);
  if (Opcao = TipoNotifacao.meEMail) then
    Email.Create().Enviar(CodCliente,Mensagem);
  if (Opcao = TipoNotifacao.meSMS) then
    SMS.Create().Enviar(CodCliente,Mensagem);
  if (Opcao = TipoNotifacao.meVoiceMail) then
    VoiceMail.Create().Enviar(CodCliente,Mensagem);
  ReadIn;
end.

// Unit do Framework de Notificação
unit GPauli.Notificacao;

interface

uses
  System.SysUtils;

type
  TipoNotifacao = (meEMail = 1, meSMS = 2, meVoiceMail = 3);

  Notificacao = class abstract
    public procedure Enviar(CodCliente: string; Mensagem: string); virtual; abstract;
  end;

  Email = class(Notificacao)
    public procedure Enviar(CodCliente: string; Mensagem: string); override;
  end;

  SMS = class(Notificacao)
    public procedure Enviar(CodCliente: string; Mensagem: string); override;
  end;

  VoiceMail = class(Notificacao)
    public procedure Enviar(CodCliente: string; Mensagem: string); override;
  end;

implementation

{ Email }

procedure Email.Enviar(CodCliente: string; Mensagem: string);
begin
  // Simula envio de um email
  WriteLn(format('Enviando EMAIL para o cliente %s',[CodCliente]));
  WriteLn(format('EMAIL: %s',[Mensagem]));
end;

{ SMS }

procedure SMS.Enviar(CodCliente: string; Mensagem: string);
begin
  // Simula envio de um SMS
  WriteLn(format('Enviando SMS para o cliente %s',[CodCliente]));
  WriteLn(format('SMS: %s',[Mensagem]));
end;

{ VoiceMail }

procedure VoiceMail.Enviar(CodCliente: string; Mensagem: string);
begin
  // Simula envio de um VoiceMail
  WriteLn(format('Enviando VoiceMail para o cliente %s',[CodCliente]));
  WriteLn(format('VoiceMail: %s',[Mensagem]));
end;

end.
```

Da mesma forma, não sobrecarrega classes descendentes com muitas responsabilidades, visto que a herança estática, assim como na vida real, é incondicional. Uma vez nascido com aquele comportamento e atributos, eles fazem parte do seu "DNA", imutável.

Outro ponto a ser observado foi o uso de um tipo enumerado para substituir os tipos strings, que obtém um tipo de domínio mais específico que pode ser verificado pelo compilador, minimizando erros e deixando o código mais claro. Executando a aplicação neste ponto, vemos que o comportamento externo observável não foi afetado, continua funcionando exatamente da mesma forma, porém a arquitetura obtida está muito mais simples de ser evoluída e mantida, mais robusta. Classes estão com responsabilidades separadas. Código está fácil de ser reutilizado. Se houver um problema com envio de e-mails, basta ir à classe que realiza essa funcionalidade e modificá-la, ou até mesmo substituí-la por outra que envie e-mails de uma melhor forma (graças ao uso de herança de abstração e polimorfismo). E finalmente, observe como ficaria simples implementar mais um tipo de envio de mensagem, caso fosse necessário, bastaria criar uma classe descendente de Notificacao (nossa abstração), sobrescrever o método polimórfico e disponibilizar essa opção no programa principal.

Refatorando para Padrões de Projeto

A refatoração melhora o design de um projeto existente. No entanto, nem todo processo de refatoração, como a aplicação de um Extract Method ou Move Method, terá como resultado uma estrutura de acordo com um determinado Padrão de Projeto (Design Pattern). Nesse caso, uma prática interessante é aplicar uma série de refatorações tendo como alvo um código seguindo uma estrutura de um padrão, como um Strategy, State, Factory ou Adapter, por exemplo. Neste caso, estamos refatorando rumo a padrões de projeto.

O código da **Listagem 7** mostra a aplicação de dois padrões de projeto em nosso exemplo, o Strategy e uma variação de Factory Method. O Strategy define uma interface para encapsular uma família de algoritmos que decidem por realizar uma operação através de polimorfismo. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizem. Esse padrão pode ser aplicado quando muitas classes têm propósitos semelhantes e divergem apenas no seu comportamento (implementação). Ou ainda, quando é necessário encapsular a solução para um problema resolvido por um ou mais algoritmos, dos quais classes clientes não devem tomar conhecimento. Este padrão traz alguns benefícios, como por exemplo, a eliminação de comandos condicionais para a seleção de um comportamento desejado (algoritmo) para resolver um problema. Ao usar herança, Strategy encapsula esses comportamentos em subclasses e usa polimorfismo (que é dinâmico) para substituir as condicionais estáticas, abrindo inclusive a arquitetura para fácil inclusão de novas estratégias para solucionar o mesmo problema. Os participantes do padrão são: i) Strategy – define uma interface comum para todos os algoritmos suportados. Context usa esta interface para chamar o algoritmo definido por uma ConcreteStrategy; ii) ConcreteStrategy – implementa o algoritmo usando a interface Strategy; iii) Context – mantém uma referência para um objeto Strategy; pode definir uma interface que permite a Strategy acessar seus dados.

O padrão Strategy resolve um dos piores bad smells, relacionados ao uso excessivo de instruções condicionais. Existem diversas refatorações destinadas à simplificação de expressões condicionais. Muitas vezes encontramos essa lógica em algoritmos, porque eles tendem a crescer, tornando-se mais sofisticado ao longo do tempo. O padrão Strategy ajuda a gerenciar a complexidade no momento em que se têm vários algoritmos destinados a resolver um determinado problema. Testar várias possibilidades com sentenças case ou if/elses aninhados tende a duplicar código. Se um novo teste for necessário na sentença, para comportar uma nova possibilidade, e esse teste for feito em vários locais, então várias alterações serão necessárias. Muitas vezes os testes podem ser substituídos por polimorfismo, que é uma das principais técnicas de programação orientada a objetos, usada internamente pelo padrão Strategy. A inclusão de muitos testes condicionais aumenta gradativamente a complexidade do código conforme o mesmo vai evoluindo ao longo do tempo, tornando sua manutenção ainda mais complexa. Quando são realizados testes unitários em código, analisar a cobertura de código afetada por testes se torna um processo trabalhoso, pois é necessário incluir lógica para testar cada uma das sentenças condicionais. E finalmente, testes condicionais dificultam a leitura e interpretação por parte de desenvolvedores que necessitam realizar manutenções corretivas no sistema de software.

Vamos a um exemplo no próprio Delphi. A classe TDataSource possui uma referência a um TDataSet, a qual existe desde o Delphi 1, portanto, adaptável a mudanças (estamos no XE6 e ela ainda existe). TDataSet é uma abstração, que possui dezenas de implementações concretas. Quando você chama o método Post, por exemplo, de qualquer tipo de TDataSet, está na verdade chamando um método virtual polimórfico. Normalmente, quando construímos formulários base de cadastro em Delphi, colocamos apenas um TDataSource em tela, mantendo TDataSets de acesso a dados separados em um TDataModule. Estes TDataSets são implementações concretas da abstração, como por exemplo, TClientDataSet (DataSnap), TADOQuery (ADO) ou TFDQuery (FireDac). Agora imagine que neste formulário base você precisa invocar o método Post para gravar os dados do formulário, de forma a tornar a implementação aberta independente de qual engine de acesso a dados utiliza. Facilmente podemos fazer uma chamada como DataSource1.DataSet.Post(). Ou seja, chamamos o método Post do DataSet que está sendo atualmente referenciado pelo TDataSource do formulário base, não importa qual tipo de TDatSet seja (TClientDataSet, TFBQuery etc.). Em tempo de execução, o método Post da correta classe concreta será invocado, através de polimorfismo, sem a necessidade de testes condicionais. Veja que neste caso, TDataSource "esconde" a estratégia de gravação usada para persistir dados de Data Controls a ele vinculados, algo muito semelhante ao que sugere o padrão Strategy. De fato, se feita uma boa arquitetura, é perfeitamente possível substituir todo um

Listagem 7. Sistema de Notificação usando Padrões de Projeto (Factory e Strategy)

```
// Programa Principal
program SistemaNotificacao;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils,
  GPauli.Notificacao in 'GPauli.Notificacao.pas';

var
  CodCliente: string;
  Mensagem: string;
  ModoEnvio: integer;
  Opcao: TipoNotifacao;
  Ctx: Context;
begin
  WriteLn('*** Sistema de notificação de clientes***');
  WriteLn('Informe código do cliente:');
  ReadLn(CodCliente);
  WriteLn('Informe mensagem a ser enviada:');
  Readln(Mensagem);
  WriteLn('Digite opção de envio, 1-EMAIL, 2-SMS, 3-Voice:');
  Readln(ModoEnvio);
  Opcao := TipoNotifacao(ModoEnvio);
  // Obtém um produto da fábrica e cria um context para strategy
  Ctx := Context.Create(FabricaNotificacao.CreateInstance(Opcao));
  Ctx.Enviar(CodCliente,Mensagem);
  Readln;
end.

// Unit do Framework de Notificação
unit GPauli.Notificacao;

interface

uses
  System.SysUtils;

type
  TipoNotifacao = (meEMail = 1, meSMS = 2, meVoiceMail = 3);

  Notificacao = class abstract
    public procedure Enviar(CodCliente: string; Mensagem: string); virtual; abstract;
  end;

  Email = class(Notificacao)
    public procedure Enviar(CodCliente: string; Mensagem: string); override;
  end;

  SMS = class(Notificacao)
    public procedure Enviar(CodCliente: string; Mensagem: string); override;
  end;

  VoiceMail = class(Notificacao)
    public procedure Enviar(CodCliente: string; Mensagem: string); override;
  end;

  Context = class
    private EstrategiaNotificacao: Notificacao;
    public procedure Enviar(CodCliente: string; Mensagem: string);
    public constructor Create(Notificacao: Notificacao);
  end;

  FabricaNotificacao = class
    public class function CreateInstance(Tipo: TipoNotifacao): Notificacao;
  end;

implementation

{ Email }

procedure Email.Enviar(CodCliente: string; Mensagem: string);
begin
  // Simula envio de um email
  WriteLn(format('Enviando EMAIL para o cliente %s',[CodCliente]));
  WriteLn(format('EMAIL: %s',[Mensagem]));
end;

{ SMS }

procedure SMS.Enviar(CodCliente: string; Mensagem: string);
begin
  // Simula envio de um SMS
  WriteLn(format('Enviando SMS para o cliente %s',[CodCliente]));
  WriteLn(format('SMS: %s',[Mensagem]));
end;

{ VoiceMail }

procedure VoiceMail.Enviar(CodCliente: string; Mensagem: string);
begin
  // Simula envio de um VoiceMail
  WriteLn(format('Enviando VoiceMail para o cliente %s',[CodCliente]));
  WriteLn(format('VoiceMail: %s',[Mensagem]));
end;

{ Context }

constructor Context.Create(Notificacao: Notificacao);
begin
  self.EstrategiaNotificacao := Notificacao;
end;

procedure Context.Enviar(CodCliente: string; Mensagem: string);
begin
  EstrategiaNotificacao.Enviar(CodCliente,Mensagem);
end;

{ FabricaNotificacao }

class function FabricaNotificacao.CreateInstance(
  Tipo: TipoNotifacao): Notificacao;
begin
  case Tipo of
    TipoNotifacao.meEMail : result := Email.Create();
    TipoNotifacao.meSMS : result := SMS.Create();
    TipoNotifacao.meVoiceMail : result := VoiceMail.Create();
  end;
end;

end.
```

engine de acesso a dados mantendo formulários intactos. Você troca de um TADOQuery configurado para persistir dados em um banco SQL Server por um TFDQuery, dessa forma, fazendo o intercâmbio dos "algoritmos", atingindo o mesmo resultado esperado (gravar no SQL Server) usando uma outra estratégia.

Em nosso caso, usamos essa estratégia para enviar uma notificação através de um dos tipos concretos criados (Email, SMS e VoiceEmail). A classe Context se encarrega de abstrair estes detalhes, recebendo uma classe base de notificação em sua construção, e através de uma chamada polimórfica delegada chama o método virtual da estratégia escolhida. A estratégia não é mais criada no programa principal, usando-se instruções condicionais. Nesse caso, um fábrica recebe por parâmetro um tipo enumerado e devolve um produto concreto (uma estratégia).

Indo além

Mais otimizações poderiam ser feitas ainda neste programa. Por exemplo, note que o método Enviar recebe tipos primitivos, e se uma mensagem precisar de mais um atributo (por exemplo, data de envio), uma reação em cadeia enorme ocorreria no framework. Nesse caso, poderíamos encapsular cada um dos parâmetros em uma classe de dados (chamada, por exemplo, de MensagemData), passando esta como referência no método Enviar. A aplicação de melhorias contínuas no código, através de técnicas de orientação a objetos, deixa sua aplicação simples, funcional, fácil de manter e de evoluir. A aplicação de padrões de projeto no contexto de refatorações segundo uma abordagem ágil evolutiva evita complexidade desnecessária, o que causaria um excesso de engenharia, em contrapartida a uma escassez de engenharia caso o sistema de software nunca fosse refatorado para um padrão. Nesse caso, refatorar para padrões de projeto de forma evolutiva se torna uma técnica vital para a aplicação

Autor

Guinther Pauli
http://www.facebook.com/guintherpauli
http://www.twitter.com/guintherpauli
http://br.linkedin.com/in/guintherpauli
http://guintherpauli.blogspot.com
http://www.devmedia.com.br/guintherpauli

Arquiteto de Software certificado pela Microsoft, Consultor, Instrutor, Autor, Palestrante, Professor de Graduação e de Pós-Graduação, Colaborador Editorial das Revistas .NET Magazine e ClubeDelphi, Certificado oficial Microsoft MCP, MCAD, MCSD.NET, MCTS, MCPD (ASP.NET, C#, MVC e Arquitetura) - Certificado oficial Delphi: 3, 5, 6, 7, 2005, 2006, Delphi para Web, Kylix, XE. Most Valuable Professional (MVP) - Embarcadero Technologies.

Links:

Refactoring - Martin Fowler
http://www.refactoring.com

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Conhecimento faz diferença!

Faça já sua assinatura digital ! | www.devmedia.com.br/es

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DEVMEDIA

FireMonkey Styles: Personalização de ListBox em aplicações mobile

Podemos observar o uso do componente ListBox nas mais diversas formas, como: menu de opções, listagem de dados, e até mesmo como organizador de layout para o formulário.

Quanto a sua forma de apresentação, podemos escolher entre algumas opções pré-definidas, através das propriedades Styles (ListBox.DefaultItemStyles.ItemStyle, ListBoxItem.StyleLookup). Porém, mais cedo ou mais tarde, iremos nos deparar com a necessidade de expor itens no ListBox com mais informações e numa disposição diferente, e provavelmente, as opções pré-definidas não atenderão a essa necessidade. Personalizar os itens do ListBox para exibir a quantidade e tipo de informações no layout desejado é algo que se torna imprescindível para qualquer programador de aplicações Mobile.

Este artigo apresentará, através de um exemplo prático, como desenvolver um modelo personalizado de TListBox-Item utilizando a tecnologia oferecida pelo framework Firemonkey no Delphi XE5.

Fique por Dentro

Este artigo é útil para quem está iniciando no desenvolvimento de aplicações Mobile utilizando Delphi XE5. Será mostrado através de um exemplo prático, como utilizar o conceito de Styles, oferecido pelo Firemonkey para personalizar itens do ListBox, um dos componentes mais utilizados neste tipo de aplicação. Tal personalização concederá ao desenvolvedor a liberdade para exibir nas listas, de acordo com o layout desejado, quantas informações acharem necessárias.

Styles

Na VCL (Biblioteca de Componentes Visuais) é possível modificar as propriedades relacionadas à aparência de cada componente no Object Inspector. Entretanto, você não vê tais propriedades look-and-feel no FireMonkey. No FireMonkey, a aparência de cada componente é definida no seu estilo, e você pode atribuir um estilo para cada componente.

Desta forma você pode facilmente mudar a aparência de todo o aplicativo, bastando apenas carregar um arquivo de estilo e associá-lo ao formulário. O Delphi XE5 oferece alguns arquivos de estilo pré-definidos, que, normalmente, são encontrados na pasta "C:\Users\Public\Documents\RAD Studio\12.0\Styles".

Este recurso pode ser comparado ao CSS (Cascading Style Sheets), linguagem de estilo utilizada em páginas HTML para aplicações web ou aos Skins utilizados pelo Delphi. Porém, muito mais do que apenas carregar arquivos com aparência para nosso projeto, o Styles nos permite manipular as características visuais e estruturais de diversos componentes. Assim podemos personalizar um componente e manter os outros com sua aparência padrão.

Styles em Aplicações Desktop x Mobile

Em aplicações FireMonkey Desktop, para carregar a tela de configuração de layout de estilos basta um clique com o botão direito do mouse sobre o componente e um clique sobre a opção "Edit Custom Style". O problema é que em aplicações Firemonkey Mobile no Delphi XE5, não encontramos estas opções, como mostra a **Figura 1**.

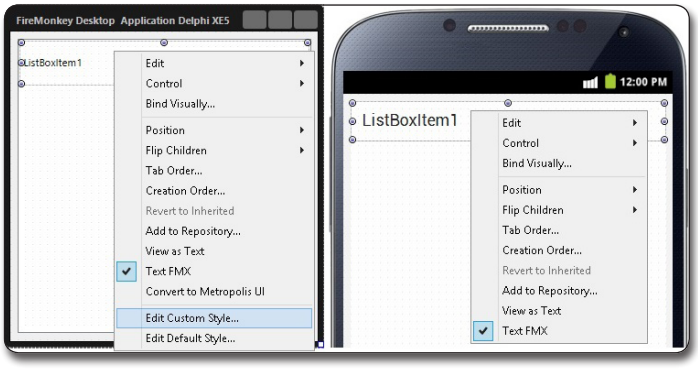


Figura 1. Menu popup aplicação desktop x menu popup em aplicação mobile

Segundo o artigo: "Can't access styles in FireMonkey mobile applications" encontrado no site da Embarcadero (veja a seção **Links**), desde a liberação do XE4, as configurações personalizadas

de estilo para aplicativos móveis foram removidas da lista de recursos, a fim de garantir compatibilidade com dispositivos iOS.

Outra opção para se ter acesso à tela de configurações de layout de estilo é adicionar ao formulário um objeto StyleBook. Um clique duplo no StyleBook abre a tela de layout de estilo, porém vazia, sem nenhuma estrutura, objetos e propriedades para customização. Devemos, então, carregar um dos arquivos de estilo disponibilizado pelo Delphi XE5. O problema é que estes arquivos de estilo acabam afetando a aparência de todos os outros componentes, enquanto que o desejado pode ser apenas personalizar um componente específico, mantendo os outros objetos no seu estado natural.

Exemplo prático: Aplicação com ListboxItem personalizado

Para demonstrar de forma prática a personalização do componente ListBox utilizando o recurso visual oferecido pelo framework Firemonkey, o Styles, desenvolvemos uma aplicação simples, com objetivo didático, que permitirá compreender o mecanismo utilizado por esta nova tecnologia. Veja nosso aplicativo em Runtime e Design time na **Figura 2**.

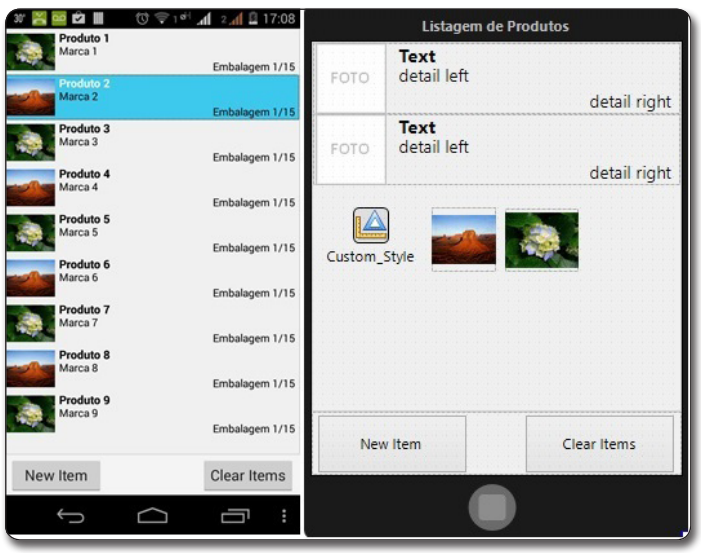


Figura 2. Aplicação para exemplo prático: Runtime (direita), e design time (esquerda)

A aplicação tem por objetivo inserir itens representando produtos numa lista. Temos um botão com rótulo New Item, que insere um novo registro a cada vez que for pressionado, e um botão com o rótulo Clear itens, que limpa todos os registros exibidos na lista. O diferencial deste exemplo está no modelo definido para os itens da lista, personalizado para esta situação e que apresenta à sua esquerda uma imagem, à sua direita, na parte superior, um texto principal em negrito e com o tamanho da fonte um pouco maior, à direita e no centro, um texto detalhe, e na parte inferior direita, um texto detalhe alinhado à direita.

A **Tabela 1** mostra os componentes que foram utilizados neste projeto, com exceção dos que serão utilizados na personalização do ListBox Item, que veremos mais adiante.

Nome	Tipo
ListBox1	TListBox
Layout1	TLayout
btnNew_Item	TButton
btnClear_Items	TButton
Image1	TImage
Image2	TImage
Custom_Style	TStyleBook

Tabela 1. Componentes do projeto

Como nosso exemplo é bastante simples e não utilizamos leitura em banco de dados, os componentes Image1 e Image2, serviram para alimentar alternadamente o campo com a imagem em nossa lista. O Componente Layout1 serviu para agrupar os botões de comando e mantê-los alinhados na parte inferior da tela.

Novo arquivo de estilo

Podemos definir um novo estilo visual às nossas aplicações Firemonkey, carregando através do componente TStyleBook um dos arquivos de estilo oferecidos pelo Delphi, e relacionando este componente ao formulário através da propriedade StyleBook. Então, todo o visual da aplicação será modificado.

Quando desejamos modificar apenas a aparência visual de um determinado componente, como é o caso deste nosso projeto de exemplo, precisamos criar um novo arquivo de estilo contendo somente as referências aos componentes desejados.

Para atender ao objetivo deste projeto criamos através do bloco de notas, um arquivo vazio e lhe demos o nome de newStyle.Style. Neste novo arquivo, adicionamos um objeto chamado TStyleContainer do tipo TLayout e extraímos do arquivo Light.Style, normalmente encontrado na pasta "C:\Users\Public\Documents\RAD Studio\12.0\Styles", os objetos com a propriedade StyleName: ListBoxStyle e ListBoxItemStyle com suas propriedades e "sub-objetos", como pode ser observado na **Figura 3**. Também é modificada a propriedade StyleName, do objeto ListBoxItemStyle para NewListBoxItemStyle. Esta mudança permite que possamos, num mesmo ListBox, exibir itens como definido pelo arquivo Light.Style e itens com nossa personalização.

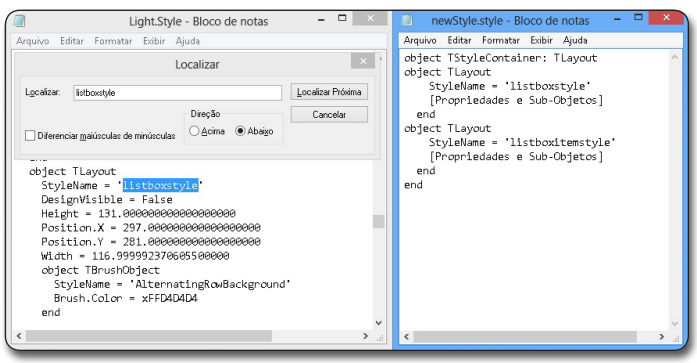


Figura 3. Localizando objetos no arquivo Light.Style e copiando para o arquivo newStyle.Style

Utilizando o novo arquivo newStyle.Style, conseguimos atender ao objetivo de modificar visualmente apenas o componente TListBox, deixando os demais com o visual em seu estado padrão. No entanto, o componente TListBox ainda apresenta as definições estabelecidas no arquivo Light.Style.

No próximo tópico, será mostrado como podemos modificar estas definições para atender ao modelo de item proposto pelo exemplo.

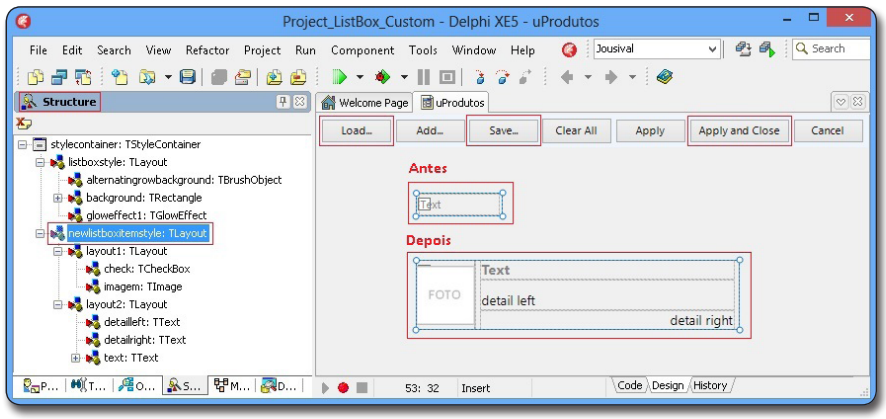


Figura 4. Tela de configuração de layout de estilos, e novo modelo visual para o ListBoxItem

Personalizando visualmente e estruturalmente o TListBoxItem

A partir de agora é abordado como foi definido visualmente e estruturalmente o item do ListBox proposto pelo projeto de exemplo.

Através de um clique duplo no componente TStyleBook temos acesso à tela de configuração de layout de estilos, onde podemos carregar o arquivo de estilo newStyle.Style, através do botão Load. A visualização da estrutura definida no arquivo pode ser obtida através da guia Structure. Observe na **Figura 4**, no quadro Antes, como era a estrutura inicial do objeto.

Para montar a nova estrutura, com NewListBoxItemStyle selecionado, adicionamos dois componentes do tipo TLayout, um do tipo TImage e mais dois TText. Observe o alinhamento dos objetos, onde Check e Image foram arrastados para layout1, e os objetos do tipo Text para layout2.

Na **Tabela 2** apresentamos melhor os componentes adicionados e suas respectivas propriedades necessárias para que o visual fique parecido com o mostrado na **Figura 4**. Para definir as propriedades de cada objeto, selecione-o através da guia Structure.

Newlistboxitemstyles	image1
Height = 62	Align = alClient
With = 300	StyleName = Imagem
Layout1	text1
Align = alLeft	Align = alClient
With = 60	HitTest = false
Layout2	HorzTextAlign = taLeading
Align = alClient	Text = detail left
Check	WordWrap = false
Align = alLeft	StyleName = detailleft
Text	text2
Align = alTop	Align = alBottom
Font->Style>fsBold = true	HitTest = false
Height = 20	HorzTextAlign = taTrailing
Margins->Todas iguais a Zero	Text = detail right
StyleName = text	WordWrap = false
	Height = 20
	StyleName = detailright

Tabela 2. Componentes e propriedades setadas no objeto NewListBoxItemStyle

Nota

A cada vez que for adicionar um componente a um objeto pela guia Structure, selecione-o primeiro.

Nota

Ao adicionar os novos componentes na estrutura do item de ListBox, caso a visualização não esteja atualizada, pressione o botão "Apply and Close" e depois retorne novamente a tela.

Após adicionados os objetos e definidas suas propriedades, salvamos todas as alterações em nosso arquivo newStyle.Style através do botão Save, exposto na tela de configuração de layout de estilos. Caso não salve as modificações, ao fazer uma nova leitura no arquivo de estilo, serão perdidas todas as configurações realizadas.

Agora, podemos utilizar este arquivo sempre que for necessário exibir o nosso item de ListBox personalizado mesmo em outros projetos.

Preenchendo a lista com o TListBoxItem personalizado

É hora de conhecer e entender a codificação utilizada nos botões de comando de nossa aplicação. Note que suas funcionalidades são bem simples. O botão New Item, a cada vez que pressionado, deve inserir um novo item na lista, e o botão Clear itens, deve excluir todos os itens da lista. A **Listagem 1** apresenta o código relacionado ao evento OnClick do botão New Item.

A linha 03 define uma variável do tipo TListBoxItem. Repare que na linha 05 criamos o objeto anteriormente declarado. Na linha seguinte, definimos a qual lista o objeto irá pertencer, neste caso a ListBox1. Já na linha 08, temos uma informação muito importante para nosso exemplo. Afinal aqui é definido qual o estilo do item. Devemos estar atentos a algumas observações, para que o item gerado possa realmente apresentar as configurações estabelecidas. Primeiramente deve-se observar a propriedade StyleBook do formulário, que deve referenciar o objeto Custom_Style. Já o objeto Custom_Style, deve ter carregado o arquivo newStyle.Style criado anteriormente.

Na linha 09 é verificado se o resultado da posição na lista é um número ímpar, caso seja afirmativo, é carregada a imagem do

objeto Image1, caso contrário, é carregada a imagem do objeto Image2. Entre as linhas 10 e 15 (com exceção da linha 13 por se tratar de um padrão), é utilizada a propriedade StyleData, para definir a informação a ser exibida em seus respectivos campos. O nome dos campos deve ser igual ao definido nas propriedades StyleName dos objetos que formam a estrutura do item personalizado. Já na **Listagem 2** temos o evento OnClick do botão Clear Items.

Listagem 1. Evento OnClick do botão New Item

```
01 procedure TfrmProdutos.btnNew_ItemClick(Sender: TObject);
02 var
03  ListBoxItem : TListBoxItem;
04 begin
05   ListBoxItem := TListBoxItem.Create(nil);
06   ListBoxItem.Parent := ListBox1;
07   ListBoxItem.Height := 55;
08   ListBoxItem.StyleLookup := 'newlistboxitemstyle';
09   if Odd(ListBoxItem.Index) then
10    ListBoxItem.StylesData['imagem'] := Image1.Bitmap
11   else
12    ListBoxItem.StylesData['imagem'] := Image2.Bitmap;
13   ListBoxItem.Text := 'Produto '+intToStr(ListBox1.Items.Count);
14   ListBoxItem.StylesData['detailleft'] := 'Marca '
      + intToStr(ListBox1.Items.Count);
15   ListBoxItem.StylesData['detailright'] := 'Embalagem 1/15';
16 end;
```

Listagem 2. Evento OnClick do botão Clear Items

```
01 procedure TfrmProdutos.btnClear_ItemsClick(Sender: TObject);
02 begin
03   ListBox1.Items.Clear;
04 end;
```

O código fonte utilizado no botão Clear Items resume-se ao comando Clear do container de itens da lista. Este comando exclui todas as linhas deixando a lista vazia.

Um pouco mais sobre Styles

A técnica apresentada pelo artigo, associada à tecnologia Styles, não se limita à personalização dos itens de uma lista. Utilizando o mesmo conceito apresentado é possível personalizar também outros componentes.

A fim de exemplificar melhor, adicionamos ao projeto de exemplo, proposto inicialmente, mais três componentes: um TEdit, um TButton e um TCheckBox, e os personalizamos, como mostrado na **Figura 5**.

Repare que ao TEdit foi adicionado um título em negrito. No componente TButton foi acrescentado um texto detalhe que ficará na parte inferior e, por último, ao TCheckBox, invertemos a posição do texto, que antes era alinhado à esquerda da caixa de seleção.

Para iniciar a personalização, é necessário copiar os objetos, EditStyle, ButtonStyle e CheckBoxsSyle do arquivo Light.Style, para o arquivo newStyle.Style (que estamos utilizando). Com os objetos no arquivo newStyle.Style, modificamos a propriedade StyleName de cada um para NewEditStyle, NewButtonStyle and NewCheckBoxStyle respectivamente.

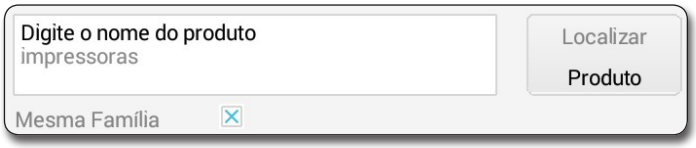


Figura 5. Componentes personalizados adicionados ao projeto de exemplo em runtime

Em seguida, através da tela de configuração de layout de estilos, é carregado novamente o arquivo newStyle.Style, clicando no botão Load. Desta forma, foi adicionado à estrutura anterior os novos objetos, como pode ser visto na **Figura 6**.

Para a personalização do objeto NewEditStyle, adicionamos um componente TText e ajustamos algumas propriedades, como pode ser visto pela **Tabela 3.** Configure-o conforme o ilustrado na mesma.

Para o objeto NewButtonStyle, também adicionamos um componente TText e ajustamos algumas propriedades, mostradas pela **Tabela 4.**

NewEditStyle
height = 36
Text (componente adicionado)
align = altop
horzTextAlign = taLeading
styleName = title
text = Título
height = 18
Margins->left = 4

Tabela 3. Componentes e propriedades do objeto NewEditStyle

NewButtonStyle
height = 42
Text (objeto já existente)
align=altop
height = 18
Text1 (componente adicionado)
align = alClient
styleName = datelhe
text = detail
height = 18

Tabela 4. Componentes e propriedades do objeto NewButtonStyle

A mudança realizada no CheckBox foi um pouco mais simples, não necessitando da inclusão de componentes. Apenas foi alterado o alinhamento do objeto layout1. A **Figura 7** mostra um passo a

passo para a configuração do CheckBox.

No tópico "Preenchendo a lista com o TLIstBoxItem personalizado" associamos o objeto ao novo estilo através de código-fonte. Para os três componentes adicionados ao exemplo, faremos este relacionamento através da propriedade StyleLookup de cada um em modo de Design. O valor atribuído à propriedade deve ser respectivamente:

- Edit1 - NewEditStyle,
- Button1 - NewButtonStyle
- CheckBox1 - NewCheckBoxStyle.

No passo 3 podemos observar a definição da propriedade StyleLookup para o componente CheckBox1.

Após aplicadas as configurações de personalização mostradas anteriormente, salvamos todas as alterações em nosso arquivo newStyle.Style através do botão Save, exposto na tela de configuração de layout de estilos.

A seguir, a **Listagem 3** mostra o código utilizado para preencher os novos campos dos objetos Edit1 e Button1 no evento OnCreate do formulário.

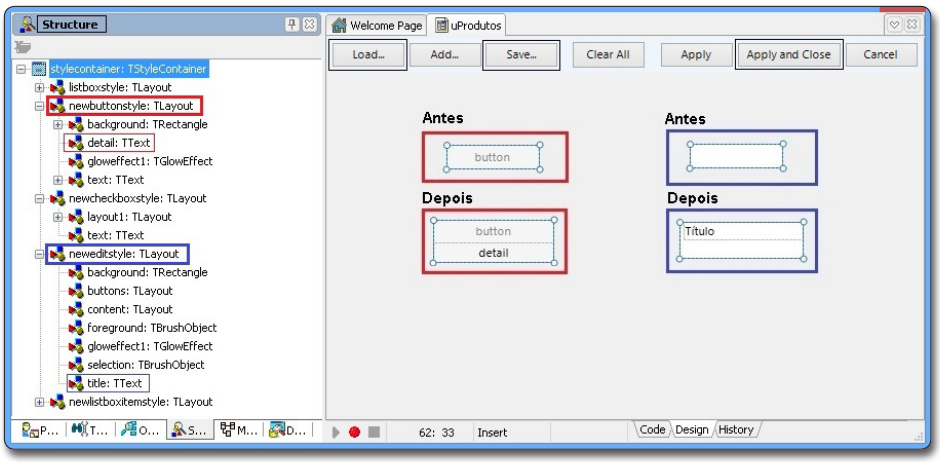


Figura 6. Tela de configuração de layout de estilos, e novo modelo visual dos novos objetos adicionados

Figura 7. Personalizando o CheckBox

Listagem 3. Evento OnCreate do formulário frmProdutos

```
01 procedure TfrmProdutos.FormCreate(Sender: TObject);
02 begin
03   Edit1.StylesData['title'] := 'Digite o nome do produto';
04   Button1.StylesData['detail'] := 'Produto';
05 end;
```

Personalizando assim esses exemplos o Firemonkey, que permite ao desenvolvedor de forma prática customização e personalização.

A utilização de listas em aplicações mobile é algo muito comum. Dificilmente veremos uma aplicação deste tipo que não faça uso de tal recurso. Este artigo apresentou, através de fundamentação teórica e exemplo prático, como personalizar listas utilizando Styles, tecnologia relativamente nova e pouco conhecida, permitindo ao desenvolvedor atender uma série de demandas que normalmente lhe serão apresentadas.

Autor

Jousival Costa Vilela
jousival@hotmail.com
Graduado em Desenvolvimento de Sistemas para Internet pela FAT Feira de Santana-BA e pós-graduado em Engenharia de Software com Enfâse em Software Livre pela UFLA Lavras-MG. Há 15 anos atuando na área de desenvolvimento de sistemas. Trabalha com Delphi desde a versão 5. Atualmente é Coordenador de TI da Tecnolens Laboratório Ótico, Grupo Essilor.

Links:

Report #: 115376 "Can't access styles in FireMonkey mobile applications"
http://qc.embarcadero.com/wc/qcmain.aspx?d=115376

Customizing the Design of a FireMonkey Application
http://docwiki.embarcadero.com/RADStudio/XE5/en/Customizing_the_Design_of_a_FireMonkey_Application

Implementing a ListView in Firemonkey
http://www.experts-exchange.com/Programming/Languages/Pascal/Delphi/A_10054-Implementing-a-ListView-in-Firemonkey.html

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!

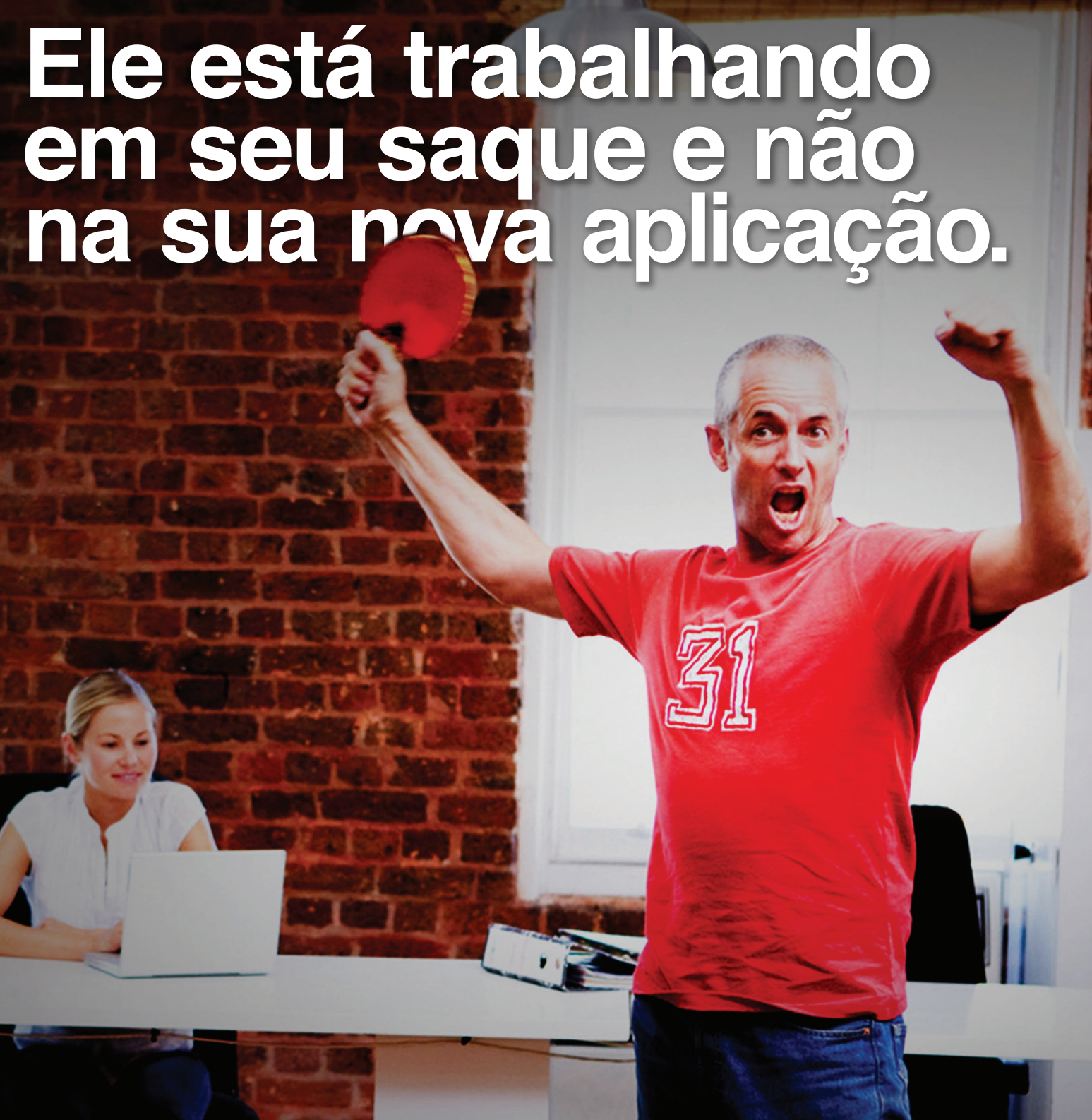
Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486

Ele está trabalhando em seu saque e não na sua nova aplicação.



Mais de 95%* dos times de desenvolvimento e testes de aplicativos relatam tempos de espera e atrasos para acessar os sistemas que necessitam para realizar suas atividades. A Virtualização de Serviços elimina estas dependências gerando ambientes simulados similares à realidade, permitindo o desenvolvimento em paralelo. Isto significa que suas aplicações serão lançadas mais rapidamente, com maior qualidade e menor custo. Game over!

Conheça mais em ca.com/br/GoDevOps

*De acordo com o estudo 2012 North America/Europe Service Virtualization Study, Coleman-Parkes.

ca technologies